Question Bank for model exam

1. What are the limitations of classic map reduce?
2. Compare classic map reduce with YARN
3. Write a note on the use of Zookeeper
4. What are the differences between HBASE and HDFS
5. List the differences Between RDBMS and Cassandra
6. Write a note on Pig latin
7. What is the use of Grunt?
8. Mention the advantages of Hive
9. Compare Relational Databases with HBase
10. When to use Hbase
11. Explain the anatomy of classic map reduce job run (16)
12. Explain in detail YARN architecture (16)
13. Explain in detail the various types of job scheduler (16)
14. Describe how the failures are handled in classic map reduce and YARN (16)
15. Explain HBASE architecture and its data model in detail (16)
16. Explain Cassandra architecture and its data model in detail (16)
17. Write a note on Cassandra clients (16)
18. Write note on Hbase clients (16)
19. Explain Hive data types and file formats (8)
20. Explain pig latin script and Grunt shell (8)
21. Describe HiveQL data definition in detail (8)

UNIT IV MAPREDUCE APPLICATIONS

MapReduce workflows – unit tests with MRUnit – test data and local tests – anatomy of MapReduce job run – classic Map-reduce – YARN – failures in classic Map-reduce and YARN – job scheduling – shuffle and sort – task execution – MapReduce types – input formats – output formats

MapReduce Workflows

It explains the data processing problem into the MapReduce model. When the processing gets more complex, the complexity is generally manifested by having more MapReduce jobs, rather than having more complex map and reduce functions. In other words, as a rule of thumb, think about adding more jobs, rather than adding complexity to jobs. Map Reduce workflow is divided into two steps:

- Decomposing a Problem into MapReduce Jobs
- Running Jobs

1. **Decomposing a Problem into MapReduce Jobs**

Let's look at an example of a more complex problem that we want to translate into a MapReduce workflow. When we write a MapReduce workflow, we have to create two scripts:

- the map script, and
- the reduce script.

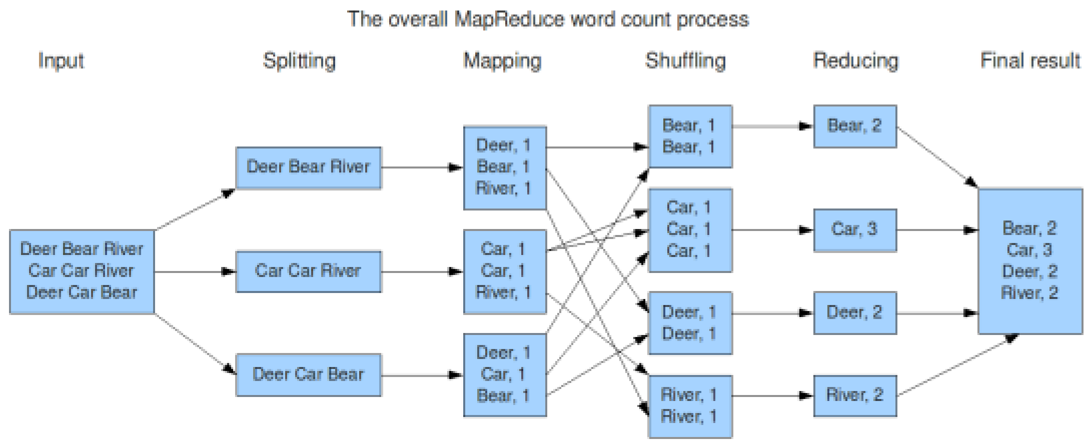
When we start a map/reduce workflow, the framework will split the input in to segments, passing each segment to a different machine. Each machine thenruns the map script on the portion of data attributed to it.

The *map script* (which you write) takes some input data, and maps it to <key, value> pairs according to your specifications. For example, if we wanted to count word frequencies in a text, we'd have <word, count> be our <key, value> pairs. Our map

script, then, would emit a <word, 1> pair for each word in the input stream. Note that the map script does no *aggregation* (i.e. actual counting) – this is what the reduce script it for. The purpose of the map script is to model the data into <key, value> pairs for the reducer to aggregate.
Emitted <key, value> pairs are then "shuffled" (to use the terminology in the diagram below), which basically means that pairs with the same key are grouped and passed to a single machine, which will then run the *reduce script* over them .

The *reduce script* (which you also write) takes a collection of <key, value> pairs and "reduces" them according to the user-specified reduce script. In our word count example, we want to count the number of word occurrences so that we can get frequencies. Thus, we'd want our reduce script to simply sum the *values* of the collection of <key, value> pairs which have the same key-.

The diagram below illustrates the described scenario nicely .



2. **Running Dependent Jobs (linear chain of jobs) or More complex Directed Acyclic Graph jobs**

When there is more than one job in a MapReduce workflow, the question arises: how do you manage the jobs so they are executed in order? There are several approaches, and the main consideration is whether you have a linear chain of jobs, or a more complex directed acyclic graph (DAG) of jobs.

For a linear chain, the simplest approach is to run each job one after another, waiting until a job completes successfully before running the next:

```
JobClient.runJob(conf1);

JobClient.runJob(conf2);
```

For anything more complex job like DAG than a linear chain, there is a class called JobControl which represents a graph of jobs to be run.

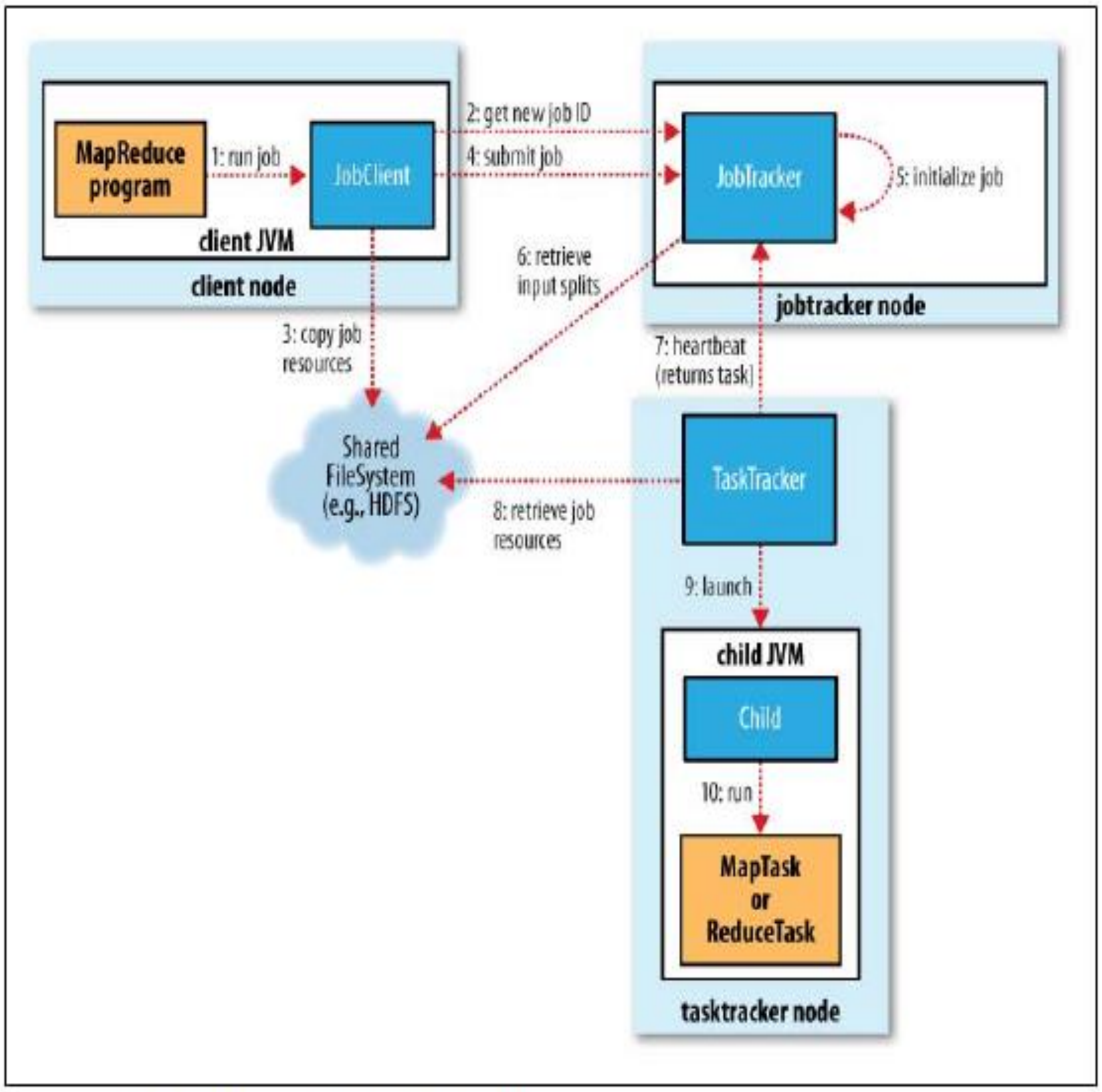
Oozie

Unlike JobControl, which runs on the client machine submitting the jobs, *Oozie* runs as a server, and a client submits a workflow to the server. In Oozie, a workflow is a DAG of *action nodes* and *control-flow nodes*. There is an action node which performs a workflow task, like moving files in HDFS, running a MapReduce job or running a Pig job. When the workflow completes, Oozie can make an HTTP callback to the client to inform it of the workflow status. It is also possible to receive callbacks every time the workflow enters or exits an action node. Oozie allows failed workflows to be re-run from an arbitrary point. This is useful for dealing with transient errors when the early actions in the workflow are time consuming to execute.

Anatomy of classic map reduce job run

How MapReduce Works? / Explain the anatomy of classic map reduce job run/How Hadoop runs map reduce Job?

You can run a MapReduce job with a single line of code: JobClient.runJob(conf). It is very short, but it conceals a great deal of processing behind the scenes. The whole process is illustrated in following figure.



How Hadoop runs a MapReduce job

As shown in Figure 1, there are four independent entities in the framework:

- Client, which submits the MapReduce Job
- JobTracker, which coordinates and controls the job run. It is a Java class called JobTracker.
- TaskerTrackers, which run the task that is split job, control the specific map or reduce task, and make reports to JobTracker. They are Java class as well.
- HDFS, which provides distributed data storage and is used to share job files between other entities.

As the Figure 1 show, a MapReduce processing including 10 steps, and in short, that is:

- The clients submit MapReduce jobs to the JobTracker.

- The JobTracker assigns Map and Reduce tasks to other nodes in the cluser
- These nodes each run a software daemon TaskTracker on separate JVM.
- Each TaskTracker actually initiates the Map or Reduce tasks and reports progress back to the JobTracker

There are six detailed levels in workflows. They are:

1. Job Submission
2. Job Initialization
3. Task Assignment
4. Task Execution
5. Task Progress and status updates
6. Task Completion

Job Submission

When the client call submit() on job object. An internal JobSubmmitter Java Object is initiated and submitJobInternal() is called. If the clients calls the waiForCompletion(), the job progresss will begin and it will response to the client with process results to clients until the job completion.
JobSubmmiter do the following work:
- Ask the JobTracker for a new job ID.
- Checks the output specification of the job.
- Computes the input splits for the job.
- Copy the resources needed to run the job. Resources include the job jar file, the configuration file and the computed input splits. These resources will be copied to HDFS in a directory named after the job id. The job jar will be copied more than 3 times across the cluster so that TaskTrackers can access it quickly.
- Tell the JobTracker that the job is ready for execution by calling submitJob() on JobTracker.

Job Initialization

When the JobTracker receives the call submitJob(), it will put the call into an internal queue from where the job scheduler will pick it up and initialize it. The initialization is done as follow:
- An job object is created to represent the job being run. It encapsulates its tasks and bookkeeping information so as to keep track the task progress and status.

- Retrieves the input splits from HDFS and create the list of tasks, each of which has task ID. JobTracker creates one map task for each split, and the number of reduce tasks according to configuration.
- JobTracker will create the setup task and cleanup task. Setup task is to create the final output directory for the job and the temporary working space for the task output. Cleanup task is to delete the temporary working space for the task ouput.
- JobTracker will assign tasks to free TaskTrackers

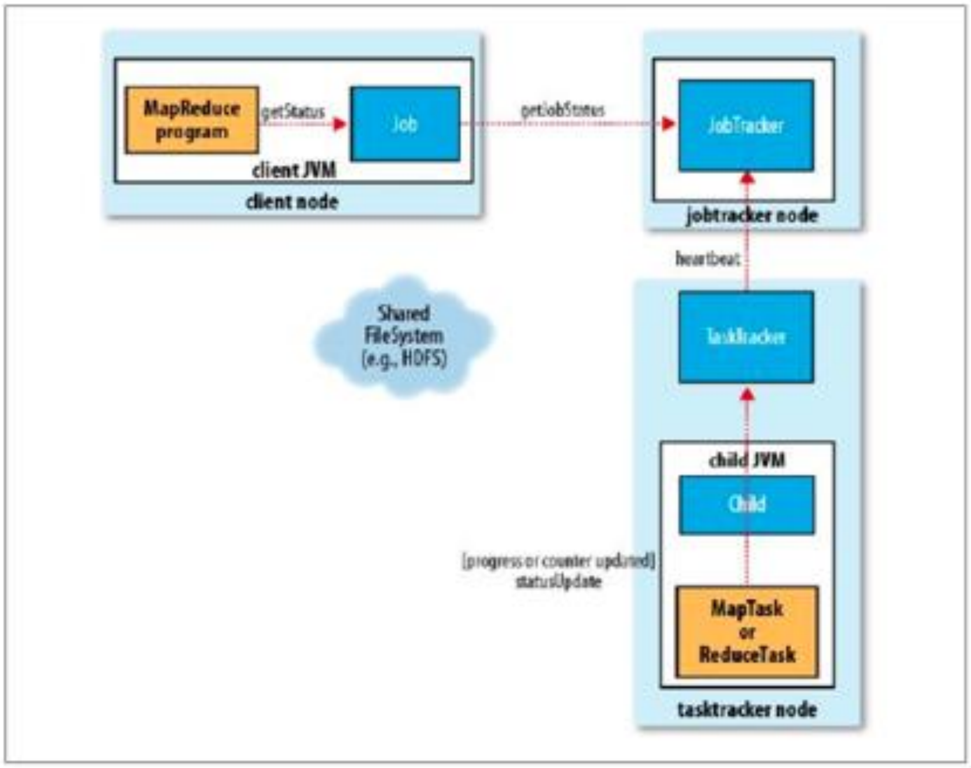
Task Assignment

TaskTrackers send heartbeat periodically to JobTracker Node to tell it if it is alive or ready to get a new task. The JobTracker will allocate a new task to the ready TaskTracker. Task assignment is as follows:
- The JobTracker will choose a job to select the task from according to scheduling algorithm, a simple way is chosen on a priority list of job. After chose the job, the JobTracker will choose a task from the job.
- TaskTrackers has a fixed number of slots for map tasks and for reduces tasks which are set independently, the scheduler will fits the empty map task slots before reduce task slots.
- To choose a reduce task, the JobTracker simply takes next in its list of yet-to-be-run reduce task, because there is no data locality consideration. But map task chosen depends on the data locality and TaskTracker's network location.

Task Execution

When the TaskTracker has been assigned a task. The task execution will be run as follows:
- Copy jar file from HDFS, copy needed files from the distributed cache on the local disk.
- Creates a local working directory for the task and 'un-jars' the jar file contents to the direcoty
- Creates a TaskRunner to run the task. The TaskRunner will lauch a new JVM to run each task.. TaskRunner fails by bugs will not affect TaskTracker. And multiple tasks on the node can reuse the JVM created by TaskRunner.
- Each task on the same JVM created by TaskRunner will run setup task and cleanup task.
- The child process created by TaskRunner will informs the parent process of the task's progress every few seconds until the task is complete.

Progress and Status Updates



After clients submit a job. The MapReduce job is a long time batching job. Hence the job progress report is important. What consists of the Hadoop task progress is as follows:

- Reading an input record in a mapper or reducer
- Writing an output record in a mapper or a reducer
- Setting the status description on a reporter, using the Reporter's setStatus() method
- Incrementing a counter
- Calling Reporter's progress()

As shown in Figure , when a task is running, the TaskTracker will notify the JobTracker its task progress by heartbeat every 5 seconds.

And mapper and reducer on the child JVM will report to TaskTracker with it's progress status every few seconds. The mapper or reducers will set a flag to indicate the status change that should be sent to the TaskTracker. The flag is checked in a separated thread every 3 seconds. If the flag sets, it will notify the TaskTracker of current task status.
The JobTracker combines all of the updates to produce a global view, and the Client can use getStatus() to get the job progress status.

Job Completion

When the JobTracker receives a report that the last task for a job is complete, it will change its status to successful. Then the JobTracker will send a HTTP notification to the client which calls the waitForCompletion(). The job statistics and the counter information will be printed to the client console. Finally the JobTracker and the TaskTracker will do clean up action for the job.

MRUnit test

MRUnit is based on JUnit and allows for the unit testing of mappers, reducers and some limited integration testing of the mapper – reducer interaction along with combiners, custom counters and partitioners.

To write your test you would:

- **Testing Mappers**

1. Instantiate an instance of the MapDriver class parameterized exactly as the mapper under test.
2. Add an instance of the Mapper you are testing in the withMapper call.
3. In the withInput call pass in your key and input value
4. Specify the expected output in the withOutput call
5. The last call runTest feeds the specified input values into the mapper and compares the actual output against the expected output set in the 'withOutput' method.

- **Testing Reducers**

1. The test starts by creating a list of objects (pairList) to be used as the input to the reducer.
2. A ReducerDriver is instantiated
3. Next we pass in an instance of the reducer we want to test in the withReducer call.
4. In the withInput call we pass in the key (of "190101") and the pairList object created at the start of the test.
5. Next we specify the output that we expect our reducer to emit
6. Finally runTest is called, which feeds our reducer the inputs specified and compares the output from the reducer against the expect output.

MRUnit testing framework is based on JUnit and it can test Map Reduce programs written on several versions of Hadoop.

Following is an example to use MRUnit to unit test a Map Reduce program that does SMS Call Deatails Record (call details record) analysis.

The records look like

1. **CDRID; CDRType; Phone1; Phone2; SMS Status Code**
655209; 1; 796764372490213; 804422938115889; 6
353415; 0; 356857119806206; 287572231184798; 4
835699; 1; 252280313968413; 889717902341635; 0

The MapReduce program analyzes these records, finds all records with CDRType as 1, and note its corresponding SMS Status Code. For example, the Mapper outputs are

6, 1
0, 1

The Reducer takes these as inputs and output number of times a particular status code has been obtained in the CDR records.

<u>The corresponding Mapper and Reducer are</u>

```
public class SMSCDRMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

  private Text status = new Text();
  private final static IntWritable addOne = new IntWritable(1);

  /**   * Returns the SMS status code and its count    */

  protected void map(LongWritable key, Text value, Context context)
                                    throws java.io.IOException, InterruptedException {

            //655209;1;796764372490213;804422938115889;6 is the Sample record format

           String[] line = value.toString().split(";");
          // If record is of SMS CDR
          if (Integer.parseInt(line[1]) == 1) {
                 status.set(line[4]);
                 context.write(status, addOne);
          }
      }
}
```

<u>The corresponding Reducer code is</u>

```
public class SMSCDRReducer extends
  Reducer<Text, IntWritable, Text, IntWritable> {

  protected void reduce(Text key, Iterable<IntWritable> values, Context context)
                                  throws java.io.IOException, InterruptedException
{
    int sum = 0;
    for (IntWritable value : values) {
      sum += value.get();
    }
    context.write(key, new IntWritable(sum));
  }
}
```

<u>The MRUnit test class for the Mapper is</u>

```
public class SMSCDRMapperReducerTest
{
   MapDriver<LongWritable, Text, Text, IntWritable> mapDriver;
  ReduceDriver<Text, IntWritable, Text, IntWritable> reduceDriver;
  MapReduceDriver<LongWritable, Text, Text, IntWritable, Text, IntWritable>
                    mapReduceDriver;

    public void setUp()
   {
    SMSCDRMapper mapper = new SMSCDRMapper();
    SMSCDRReducer reducer = new SMSCDRReducer();
    mapDriver = MapDriver.newMapDriver(mapper);;
    reduceDriver = ReduceDriver.newReduceDriver(reducer);
    mapReduceDriver = MapReduceDriver.newMapReduceDriver(mapper, reducer);
  }

  @Test
  public void testMapper()
 {
    mapDriver.withInput(new LongWritable(), new Text(  "655209;1;796764372490213;8044229
                                                          38115889;6"));
    mapDriver.withOutput(new Text("6"), new IntWritable(1));
    mapDriver.runTest();
  }
 @Test
 public void testReducer()
{
   List<IntWritable> values = new ArrayList<IntWritable>();
   values.add(new IntWritable(1));
   values.add(new IntWritable(1));
   reduceDriver.withInput(new Text("6"), values);
   reduceDriver.withOutput(new Text("6"), new IntWritable(2));
   reduceDriver.runTest();
 }
}
```

YARN: It is a Hadoop MapReduce 2 and developed to address the various limitations of classic map reduce

Current MapReduce (classic) Limitations:

- Scalability problem
    - Maximum Cluster Size – 4000 Nodes only
    - Maximum Concurrent Tasks – 40000 only
    - Coarse synchronization in Job Tracker
- It supports Single point of failure
    - When a failure occurs it kills all queued and running jobs
    - Jobs need to be resubmitted by users
- Restart is very tricky due to complex state

For large clusters with more than 4000 nodes, the classic MapReduce framework hit the scalability problems.

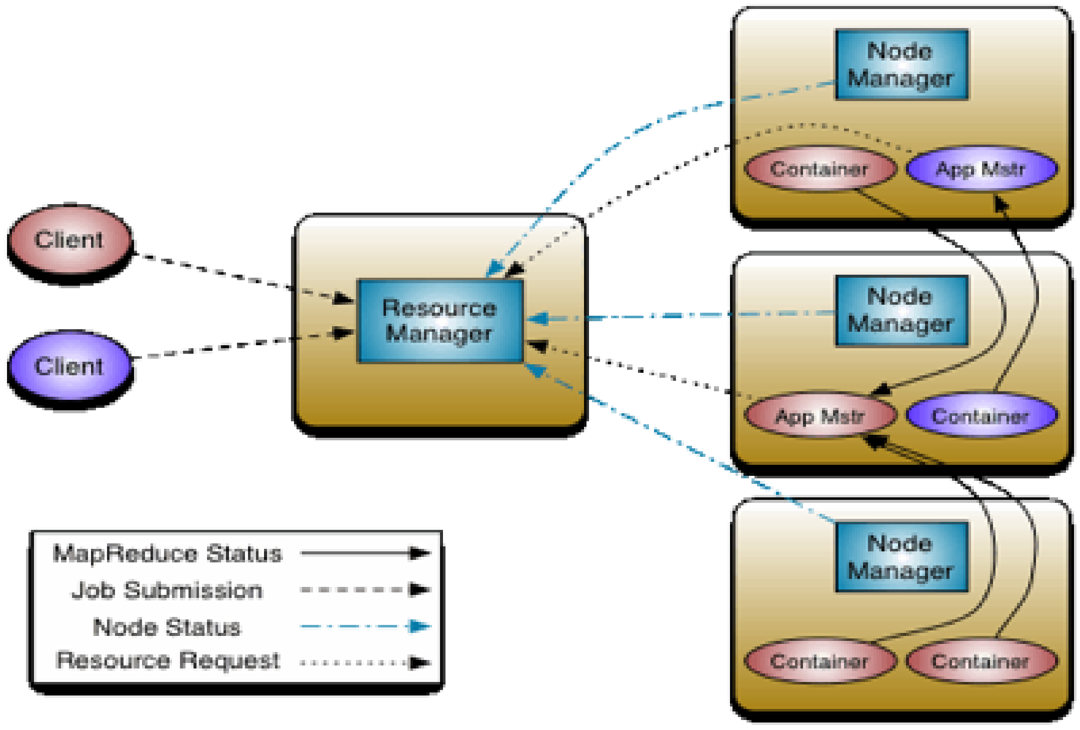
YARN stands for **Yet Another Resource Negotiator**

A group in Yahoo began to design the next generation MapReduce in 2010, and in 2013 Hadoop 2.x releases MapReduce 2, Yet Another Resource Negotiator (YARN) to remedy the sociability shortcoming.

What does Yarn do ?

- ❑ Provides a cluster level resource manager
- ❑ Adds application level resource management
- ❑ Provides slots for jobs other than Map / Reduce
- ❑ Improves resource utilization
- ❑ Improves scaling
    - ❑ Cluster size is 6000 – 10000 nodes
    - ❑ 100,000+ concurrent tasks can be executed
    - ❑ 10,000 concurrent jobs can be executed
- ❑ Split Job Tracker into
    1. Resource Manager(RM): performs cluster level resource management
    2. Application Master(AM): performs job Scheduling and Monitoring

YARN Architecture



As shown in Figure, the YARN involves more entities than classic MapReduce 1 :
- Client, the same as classic MapReduce which submits the MapReduce job.
- Resource Manager, which has the ultimate authority that arbitrates resources among all the applications in the cluster, it coordinates the allocation of compute resources on the cluster.
- Node Manager, which is in charge of resource containers, monitoring resource usage (cpu, memory, disk , network) on the node , and reporting to the Resource Manager.
- Application Master, which is in charge of the life cycle an application, like a MapReduce Job. It will negotiates with the Resource Manager of cluster resources—in YARN called containers. The Application Master and the MapReduce task in the containers are scheduled by the Resource Manager. And both of them are managed by the Node Manager. Application Mater is also responsible for keeping track of task progress and status.
- HDFS, the same as classic MapReduce, for files sharing between different entities.

Resource Manager consists of two components:

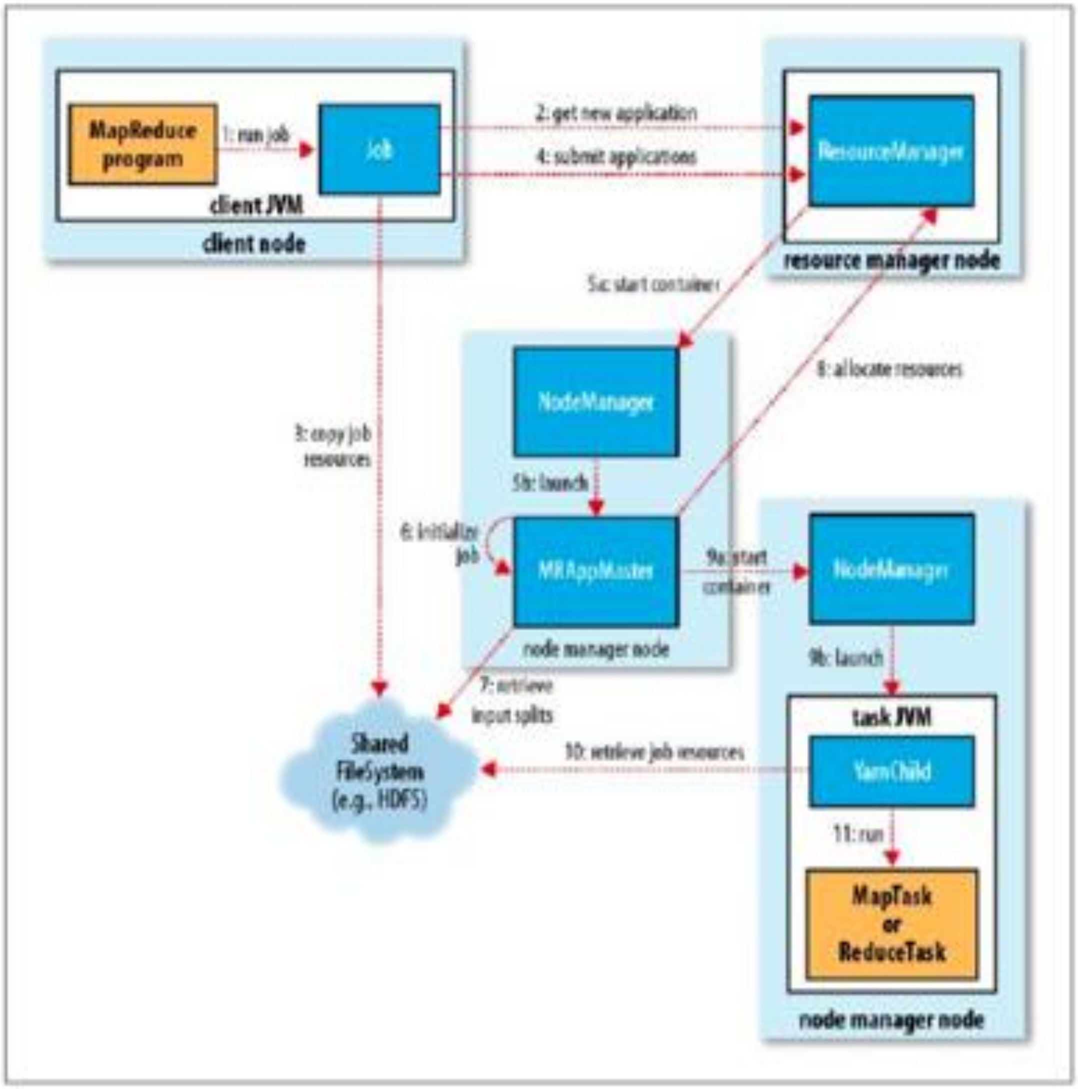
- Scheduler and
- Applications Manager.

Scheduler is in charge of allocating resources. The resource Container incorporates elements such as memory, cup, disk, network etc. Scheduler just has the resource allocation function, has no responsible for job status monitoring. And the scheduler is pluggable, can be replaced by other scheduler plugin-in.

The ApplicationsManager is responsible for accepting job-submissions, negotiating the first container for executing the application specific Application Master, and it provides restart service when the container fails.

The MapReduce job is just one type of application in YARN. Different application can run on the same cluster with YARN framework.

YARN MapReduce



As shown in above Figure, it is the MapReduce process with YARN, there are 11 steps, and we will explain it in 6 steps the same as the MapReduce 1 framework. They are Job Submission, Job Initialization, Task Assignment, Task Execution, Progress and Status Updates, and Job Completion.

Job Submission

Clients can submit jobs with the same API as MapReduce 1 in YARN. YARN implements its ClientProtocol, the submission process is similar to MapReduce 1.
- The client calls the submit() method, which will initiate the JobSubmmitter object and call submitJobInternel().
- Resource Manager will allocate a new application ID and response it to client.
- The job client checks the output specification of the job
- The job client computes the input splits
- The job client copies resources, including the splits data, configuration information, the job JAR into HDFS
- Finally, the job client notify Resource Manager it is ready by calling submitApplication() on the Resource Manager.

Job Initialization

When the Resource Manager(RM) receives the call submitApplication(), RM will hands off the job to its scheduler. The job initialization is as follows:
- The scheduler allocates a resource container for the job,
- The RM launches the Application Master under the Node Manager's management.
- Application Master initialize the job. Application Master is a Java class named MRAppMaster, which initializes the job by creating a number of bookkeeping objects to keep track of the job progress. It will receive the progress and the completion reports from the tasks.
- Application Master retrieves the input splits from HDFS, and creates a map task object for each split. It will create a number of reduce task objects determined by the mapreduce.job.reduces configuration property.
- Application Master then decides how to run the job.

For small job, called uber job, which is the one has less than 10 mappers and only one reducer, or the input split size is smaller than a HDFS block, the Application Manager will run the job on its own JVM sequentially. This policy is different from MapReduce 1 which will ignore the small jobs on a single TaskTracker.

For large job, the Application Master will launches a new node with new NodeManager and new container, in which run the task. This can run job in parallel and gain more performance.

Application Master calls the job setup method to create the job's output directory. That's different from MapReduce 1, where the setup task is called by each task's TaskTracker.

Task Assignment

When the job is very large so that it can't be run on the same node as the Application Master. The Application Master will make request to the Resource Manager to negotiate more resource container which is in piggybacked on heartbeat calls. The task assignment is as follows:
- The Application Master make request to the Resource Manager in heartbeat call. The request includes the data locality information, like hosts and corresponding racks that the input splits resides on.
- The Recourse Manager hand over the request to the Scheduler. The Scheduler makes decisions based on these information. It attempts to place the task as close the data as possible. The data-local nodes is great, if this is not possible , the rack-local the preferred to nolocal node.
- The request also specific the memory requirements, which is between the minimum allocation (1GB by default) and the maximum allocation (10GB). The Scheduler will schedule a container with multiples of 1GB memory to the task, based on the mapreduce.map.memory.mb and mapreduce.reduce.memory.mb property set by the task.

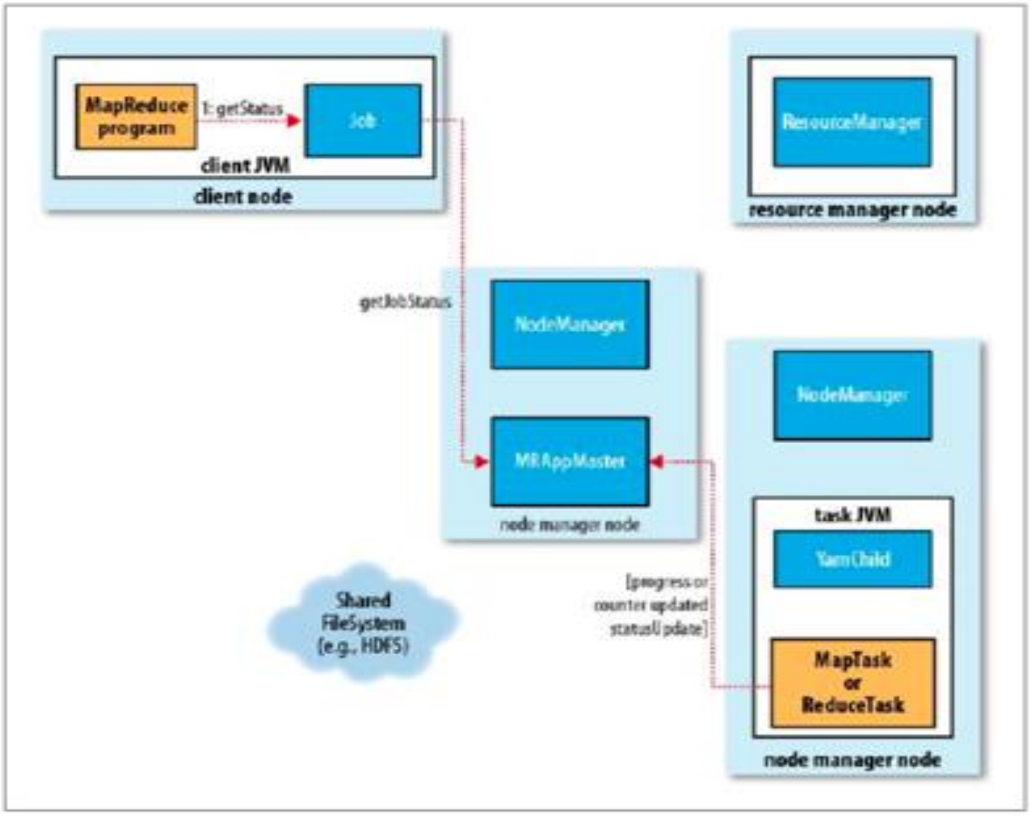
This way is more flexible than MapReduce 1. In MapReduce 1, the TaskTrackers have a fixed number of slots and each task runs in a slot. Each slot has fixed memory allowance which results in two problems. For small task, it will waste of memory, and for large task which need more memeory, it will lack of memory. In YARN, the memory allocation is more fine-grained, which is also the beauty of YARE resides in.

Task Execution

After the task has been assigned the container by the Resource Manger's scheduler, the Application Master will contact the NodeManger which will launch the task JVM.
The task execution is as follows:
- The Java Application whose class name is YarnChild localizes the resources that the task needs. YarnChild retrieves job resources including the job jar, configuration file, and any needed files from the HDFS and the distributed cache on the local disk.
- YarnChild run the map or the reduce task
Each YarnChild runs on a dedicated JVM, which isolates user code from the long running system daemons like NodeManager and the Application Master. Different from MapReduce 1, **YARN doesn't support JVM reuse**, hence each task must run on new JVM.
The streaming and the pipeline processs and communication in the same as MapReduce 1.

Progress and Status Updates

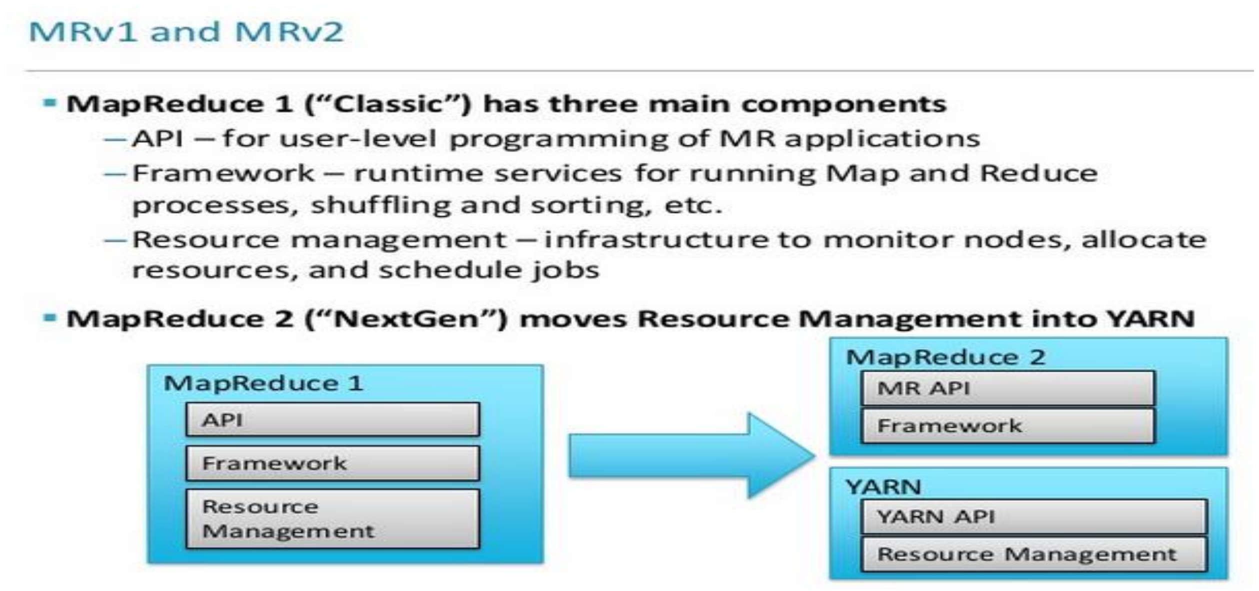
When the job is running under YARN, the mapper or reducer will report its status and progress to its Application Master every 3 seconds over the umbilical interface. The Application Master will aggregate these status reports into a view of the task status and progress. While in MapReduce 1, the TaskTracker reports status to JobTracker which is responsible for aggregating status into a global view.
Moreover, the Node Manger will send heartbeats to the Resource Manager every few seconds. The Node Manager will monitoring the Application Master and the recourse container usage like cpu, memeory and network, and make reports to the Resource Manager. When the Node Manager fails and stops heartbeat the Resource Manager, the Resource Manager will remove the node from its available resource nodes pool.
The client pulls the status by calling getStatus() every 1 second to receive the progress updates, which are printed on the user console. User can also check the status from the web UI. The Resource Manager web UI will display all the running applications with links to the web UI where displays task status and progress in detail.



Job Completion

Every 5 second the client will check the job completion over the HTTP ClientProtocol by calling waitForCompletion(). When the job is done, the Application Master and the task containers clean up their working state and the outputCommitter's job cleanup method is called. And the job information is archived as history for later interrogation by user.

Compare Classic Map Reduce with YARN map reduce



MRv1 and MRv2

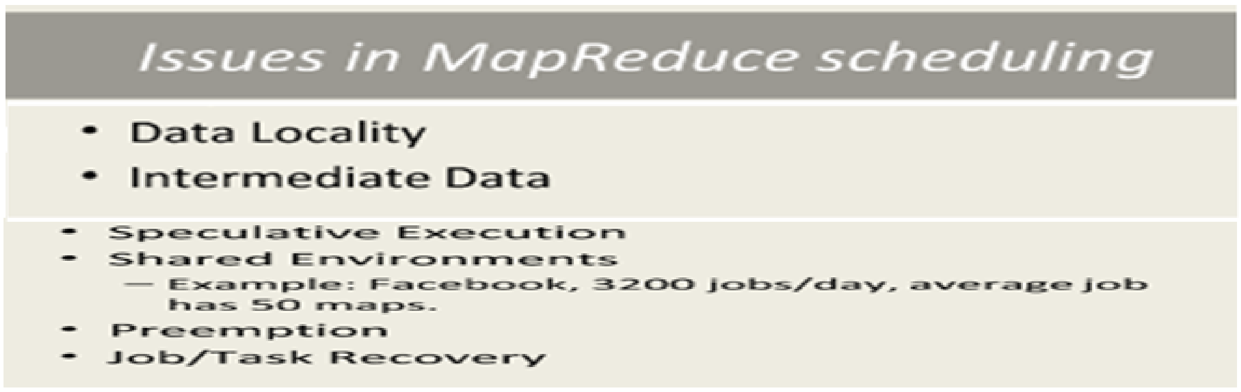
- **MapReduce 1 ("Classic") has three main components**
  - API – for user-level programming of MR applications
  - Framework – runtime services for running Map and Reduce processes, shuffling and sorting, etc.
  - Resource management – infrastructure to monitor nodes, allocate resources, and schedule jobs
- **MapReduce 2 ("NextGen") moves Resource Management into YARN**

MapReduce 1
API
Framework
Resource Management
MapReduce 2
MR API
Framework
YARN
YARN API
Resource Management

- ❖ YARN has Fault Tolerance(continue to work in the event of failure) and Availability
  - ❖ Resource Manager
    - ❖ No single point of failure – state saved in ZooKeeper
    - ❖ Application Masters are restarted automatically on RM restart
  - ❖ Application Master
    - ❖ Optional failover via application-specific checkpoint
    - ❖ MapReduce applications pick up where they left off via state saved in HDFS
- ❖ YARN has Network Compatibility
  - ❖ Protocols are wire-compatible

    - Old clients can talk to new servers
    - Rolling upgrades
- YARN supports for programming paradigms other than MapReduce (Multi tenancy)
    - Tez – Generic framework to run a complex MR
    - HBase on YARN
    - Machine Learning: Spark
    - Graph processing: Giraph
    - Real-time processing: Storm
- YARN is Enabled by allowing the use of paradigm-specific application master
- YARN runs all on the same Hadoop cluster
- YARN's biggest advantage is multi-tenancy, being able to run multiple paradigms simultaneously is a big plus.

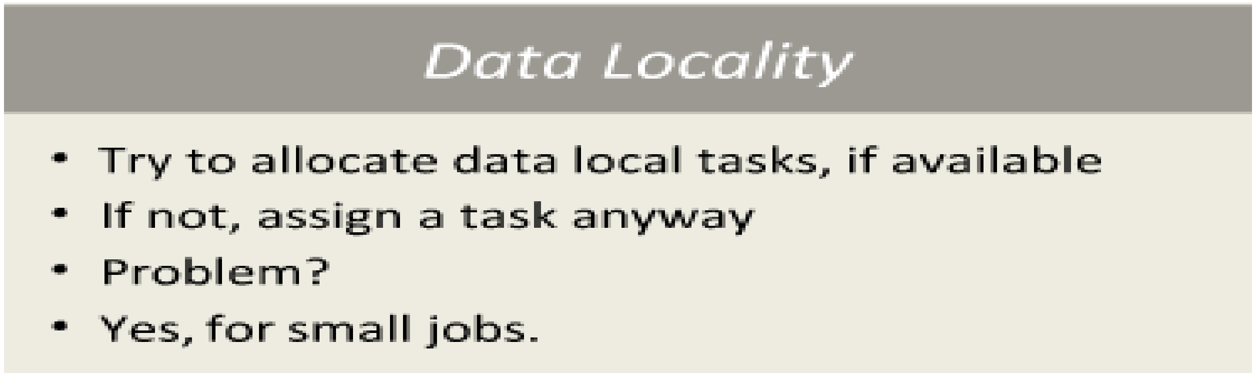
Job Scheduling in MapReduce



Issues in MapReduce scheduling

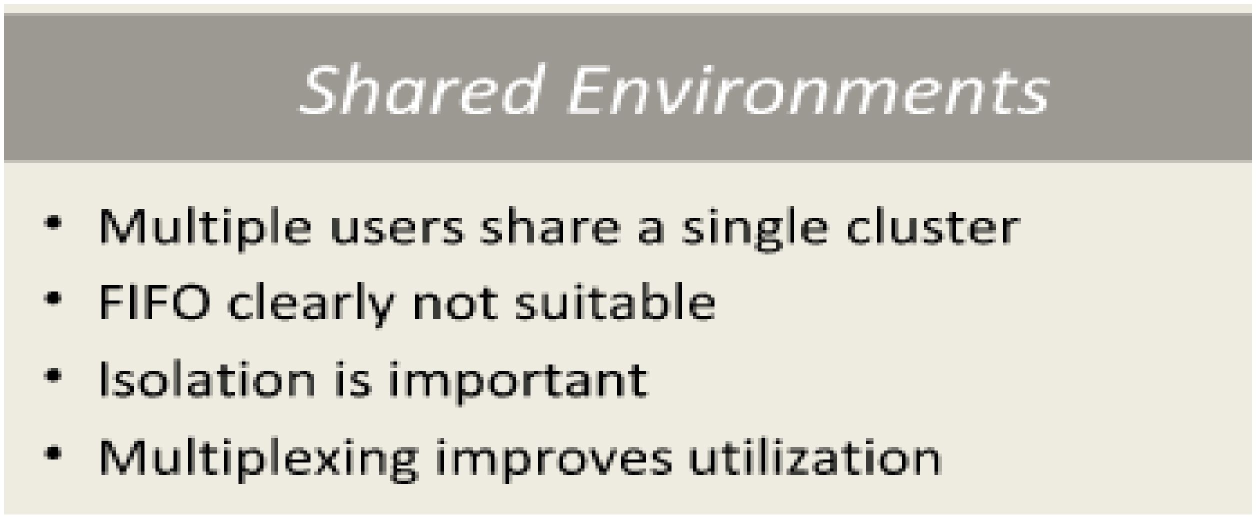
- Data Locality
- Intermediate Data
- Speculative Execution
- Shared Environments
    - Example: Facebook, 3200 jobs/day, average job has 50 maps.
- Preemption
- Job/Task Recovery

Speculative Execution: One problem with the Hadoop system is that by dividing the tasks across many nodes, it is possible for a few slow nodes to limit the rest of the program.
Preemption: task can be allowed to suspend and resumed



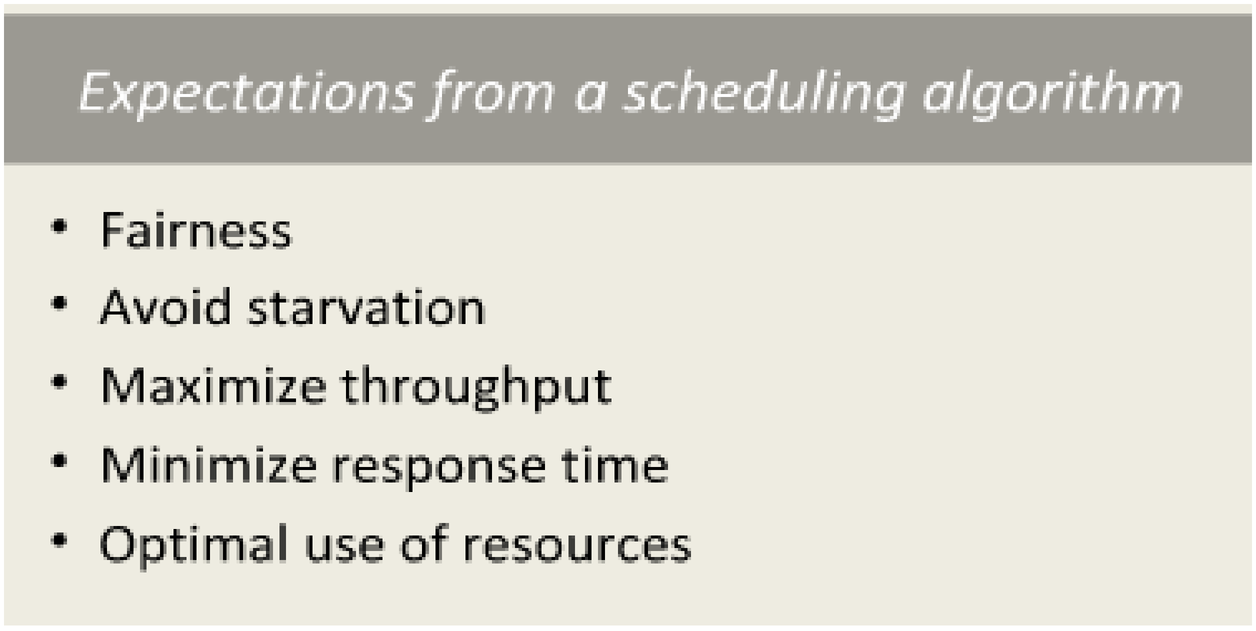
Data Locality

- Try to allocate data local tasks, if available
- If not, assign a task anyway
- Problem?
- Yes, for small jobs.



Shared Environments

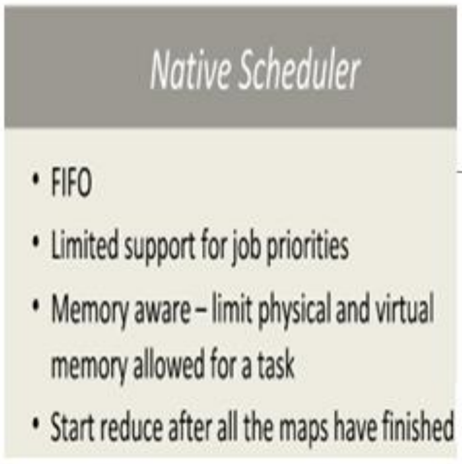
- Multiple users share a single cluster
- FIFO clearly not suitable
- Isolation is important
- Multiplexing improves utilization



Expectations from a scheduling algorithm

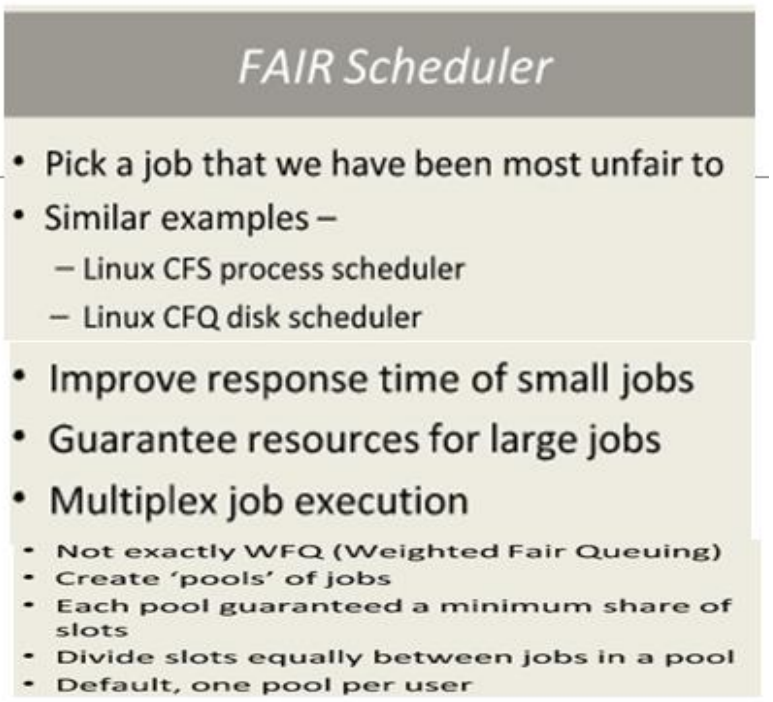
- Fairness
- Avoid starvation
- Maximize throughput
- Minimize response time
- Optimal use of resources

Types of job schedulers



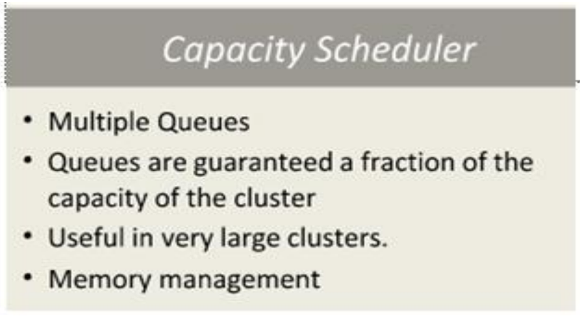
Native Scheduler

- FIFO
- Limited support for job priorities
- Memory aware – limit physical and virtual memory allowed for a task
- Start reduce after all the maps have finished



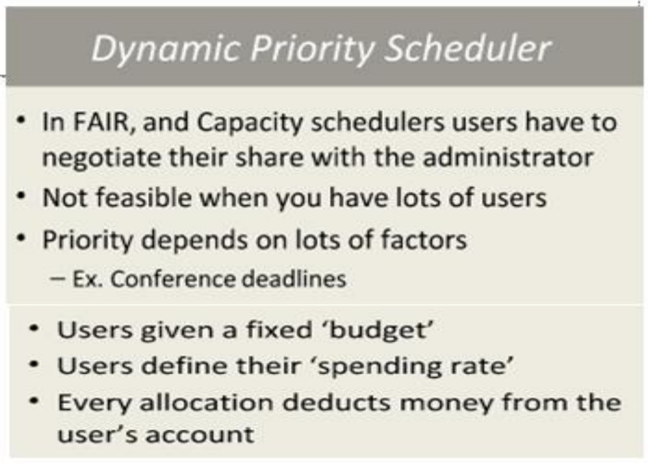
FAIR Scheduler

- Pick a job that we have been most unfair to
- Similar examples –
  - Linux CFS process scheduler
  - Linux CFQ disk scheduler
- Improve response time of small jobs
- Guarantee resources for large jobs
- Multiplex job execution
- Not exactly WFQ (Weighted Fair Queuing)
- Create 'pools' of jobs
- Each pool guaranteed a minimum share of slots
- Divide slots equally between jobs in a pool
- Default, one pool per user



Capacity Scheduler

- Multiple Queues
- Queues are guaranteed a fraction of the capacity of the cluster
- Useful in very large clusters.
- Memory management



Dynamic Priority Scheduler

- In FAIR, and Capacity schedulers users have to negotiate their share with the administrator
- Not feasible when you have lots of users
- Priority depends on lots of factors
  - Ex. Conference deadlines
- Users given a fixed 'budget'
- Users define their 'spending rate'
- Every allocation deducts money from the user's account

Failures in classic map reduce

One of the major benefits of using Hadoop is its ability to handle such failures and allow your job to complete.

1. **Task Failure**

- Consider first the case of the child task failing. The most common way that this happens is when user code in the map or reduce task throws a runtime exception. If this happens, the child JVM reports the error back to its parent tasktracker, before it exits. The error ultimately makes it into the user logs. The tasktracker marks the task attempt as *failed*, freeing up a slot to run another task.

- For Streaming tasks, if the Streaming process exits with a nonzero exit code, it is marked as failed. This behavior is managed by the stream.non.zero.exit.is.failure property.
- Another failure mode is the sudden exit of the child JVM. In this case, the tasktracker notices that the process has exited and marks the attempt as failed.
- A task attempt may also be *killed*, which is also different kind of failing. Killed task attempts do not count against the number of attempts to run the task since it wasn't the task's fault that an attempt was killed.

2. Tasktracker Failure

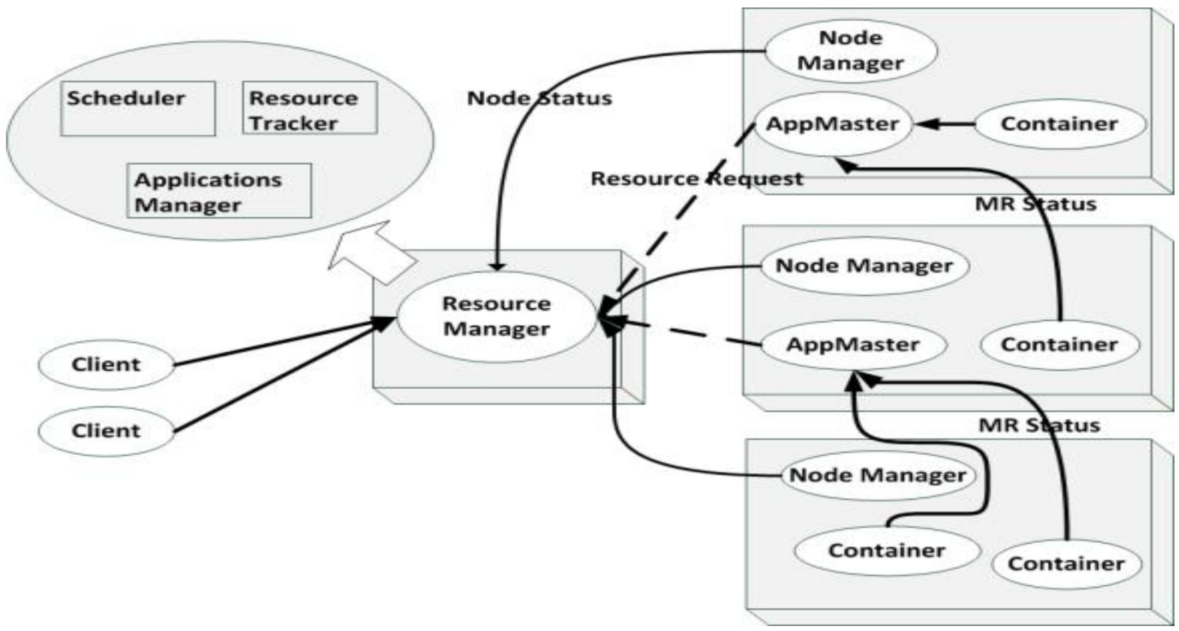
- If a tasktracker fails by crashing, or running very slowly, it will stop sending heartbeats to the jobtracker (or send them very infrequently). The jobtracker will notice a tasktracker that has stopped sending heartbeats (if it hasn't received one for 10 minutes) and remove it from its pool of tasktrackers to schedule tasks on.

- A tasktracker can also be *blacklisted* by the jobtracker, even if the tasktracker has not failed. A tasktracker is blacklisted if the number of tasks that have failed on it is significantly higher than the average task failure rate on the cluster. Blacklisted tasktrackers can be restarted to remove them from the jobtracker's blacklist.

3. Jobtracker Failure

- Failure of the jobtracker is the most serious failure mode. Currently, Hadoop has no mechanism for dealing with failure of the jobtracker—it is a single point of failure— so in this case the job fails.

Failures in YARN



Refer to yarn architecture figure above**, container and task failures are handled by node-manager**. When a container fails or dies, node-manager detects the failure event and launches a new container to replace the failing container and restart the task execution in the new container. In the event of application-master failure, the resource-manager detects the failure and start a new instance of the application-master with a new container. The ability to recover the associated job state depends on the application-master implementation. MapReduce application-master has the ability to recover the state but it is not enabled by default. Other than resource-manager, associated client also reacts with the failure. The client contacts the resource-manager to locate the new application-master's address.

Upon failure of a node-manager, the resource-manager updates its list of available node-managers. Application-master should recover the tasks run on the failing node-managers but it depends on the application-master implementation. MapReduce application-master has an additional capability to recover the failing task and blacklist the node-managers that often fail.

Failure of the resource-manager is severe since clients can not submit a new job and existing running jobs could not negotiate and request for new container. Existing node-managers and application-masters try to reconnect to the failed resource-manager. The job progress will be lost when they are unable to reconnect. This lost of job progress will likely frustrate engineers or data scientists that use YARN because typical production jobs that run on top of YARN are expected to have long running time and typically they are in the order of few hours. Furthermore, this limitation is preventing YARN to be used efficiently in cloud environment (such as Amazon EC2) since node failures often happen in cloud environment.

Shuffle and Sort

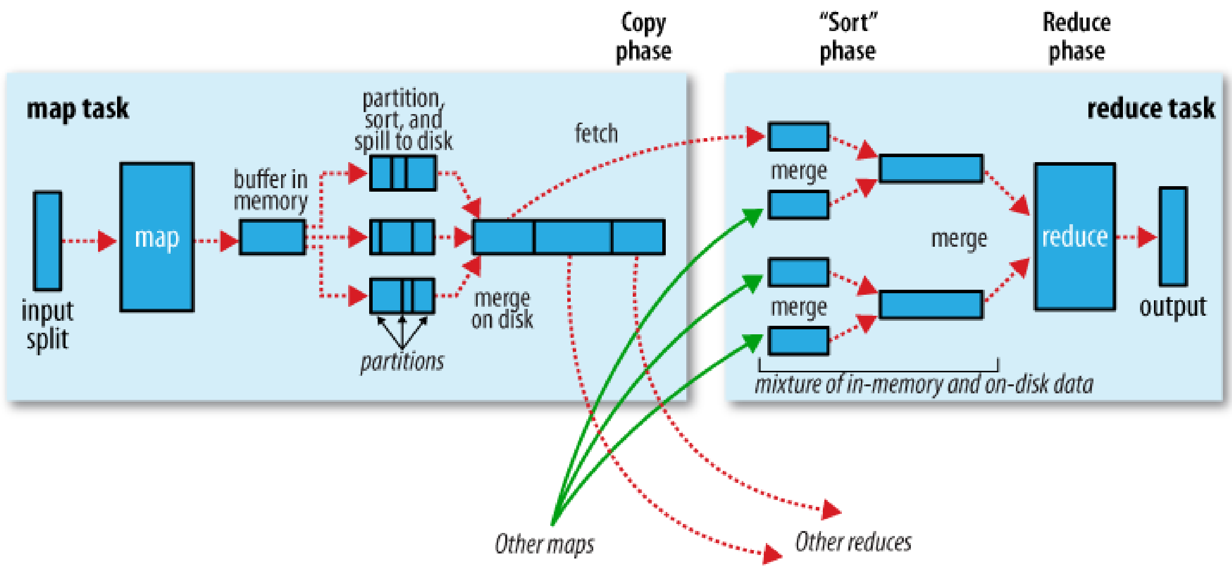
- MapReduce makes the guarantee that the input to every reducer is sorted by key.
- The process by which the system performs the sort—and transfers the map outputs to the reducers as inputs—is known as the shuffle.
- The shuffle is an area of the codebase where refinements and improvements are continually being made.

STEPS

1. The Map Side
2. The Reduce Side
3. Configuration Tuning

I. The Map Side

When the map function starts producing output, it is not simply written to disk. The process is more involved, and takes advantage of buffering writes in memory and doing some presorting for efficiency reasons.



Shuffle and sort in MapReduce

The buffer is 100 MB by default, a size which can be tuned by changing the io.sort.mb property. When the contents of the buffer reaches a certain threshold size a background thread will start to spill the contents to disk. Map outputs will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time, the map will block until the spill is complete. Spills are written in round-robin fashion to the directories specified by the mapred.local.dir property, in a job-specific subdirectory. Before it writes to disk, the thread first divides the data into partitions corresponding to the reducers that they will ultimately be sent to. Within each partition, the background thread performs an in-memory sort by key, and if there is a combiner function, it is run on the output of the sort.

Running the combiner function makes for a more compact map output, so there is less data to write to local disk and to transfer to the reducer. Each time the memory buffer reaches the spill threshold, a new spill file is created, so after the map task has written its last output record there could be several spill files. Before the task is finished, the spill files are merged into a single partitioned and sorted output file. The configuration property io.sort.factor controls the maximum number of streams to merge at once; the default is 10.If there are at least three spill files then the combiner is run again before the output file is written. Combiners may be run repeatedly over the input without affecting the final result.If there are only one or two spills, then the potential reduction in map output size is not worth the overhead in invoking the combiner, so it is not run again for this map output.

To compress the map output as it is written to disk, makes it faster to write to disk, saves disk space, and reduces the amount of data to transfer to the reducer. By default, the output is not compressed, but it is easy to enable by setting mapred.compress.map.output to true. The output file's partitions are made available to the reducers over HTTP. The maximum number of worker threads used to serve the file partitions is controlled by the tasktracker.http.threads property. The default of 40 may need increasing for large clusters running large jobs.

II. The Reduce Side

The map output file is sitting on the local disk of the machine that ran the map task. The reduce task needs the map output for its particular partition from several map tasks across the cluster.

Copy phase of reduce: The map tasks may finish at different times, so the reduce task starts copyingtheir outputs as soon as each completes. The reduce task has a small number of copier threads so that it can fetch map outputs in parallel.

The default is five threads, but this number can be changed by setting the mapred.reduce.parallel.copies property. The map outputs are copied to reduce task JVM's memory otherwise, they are copied to disk. When the in-memory buffer reaches a threshold size or reaches a threshold number of map outputs it is merged and spilled to disk. If a combiner is specified it will be run during the merge to reduce the amount of data written to disk.The copies accumulate on disk, a background thread merges them into larger, sorted files. This saves some time merging later on.

Any map outputs that were compressed have to be decompressed in memory in order to perform a merge on them. When all the map outputs have been copied, the reduce task moves into the sort phase which merges the map outputs, maintaining their sort ordering. This is done in rounds. For example, if there were 50 map outputs, and the merge factor was 10, then there would be 5 rounds. Each round would merge 10 files into one, so at the end there would be five intermediate files. These five files into a single sorted file, the merge saves a trip to disk by directly feeding the reduce function. This final merge can come from a mixture of in-memory and on-disk segments.

During the reduce phase, the reduce function is invoked for each key in the sorted output. The output of this phase is written directly to the output filesystem, typically HDFS. In the case of HDFS, since the tasktracker node is also running a datanode, the first block replica will be written to the local disk.

III. Configuration Tuning

To understand how to tune the shuffle to improve MapReduce performance. The general principle is to give the shuffle as much memory as possible. There is a trade-off, in that you need to make sure that your map and reduce functions get enough memory to operate. Write your map and reduce functions to use as little memory as possible. They should not use an unbounded amount of memory. The amount of memory given to the JVMs in which the map and reduce tasks run is set by the mapred.child.java.opts property. To make this as large as possible for the amount of memory on your task nodes.

On the map side, the best performance can be obtained by avoiding multiple spills to disk; one is optimal. If you can estimate the size of your map outputs, then you can set the io.sort.* properties appropriately to minimize the number of spills. There is a MapReduce counter that counts the total number of records that were spilled to disk over the course of a job, which can be useful for tuning. The counter includes both map and reduces side spills.

On the reduce side, the best performance is obtained when the intermediate data can reside entirely in memory. By default, this does not happen, since for the general case all the memory is reserved for the reduce function. But if your reduce function has light memory requirements, then setting mapred.inmem.merge.threshold to 0 and mapred.job.reduce.input.buffer.percent to 1.0 may bring a performance boost. Hadoop uses a buffer size of 4 KB by default, which is low, so you should increase this across the cluster.

Input Formats

Hadoop can process many different types of data formats, from flat text files to databases.

1) Input Splits and Records:

An input split is a chunk of the input that is processed by a single map. Each map processes a single split. Each split is divided into records, and the map processes each record—a key-value pair—in turn.

```
public abstract class InputSplit {
public abstract long getLength() throws IOException, InterruptedException;
public abstract String[] getLocations() throws IOException, InterruptedException;
}
```

- FileInputFormat: FileInputFormat is the base class for all implementations of InputFormat that use files as their data source. It provides two things: a place to define which files are included as the input to a job, and an implementation for generating splits for the input files.
- FileInputFormat input paths: The input to a job is specified as a collection of paths, which offers great flexibility in constraining the input to a job. FileInputFormat offers four static convenience methods for setting a Job's input paths:

```
public static void addInputPath(Job job, Path path)
public static void addInputPaths(Job job, String commaSeparatedPaths)
public static void setInputPaths(Job job, Path... inputPaths)
public static void setInputPaths(Job job, String commaSeparatedPaths)
```

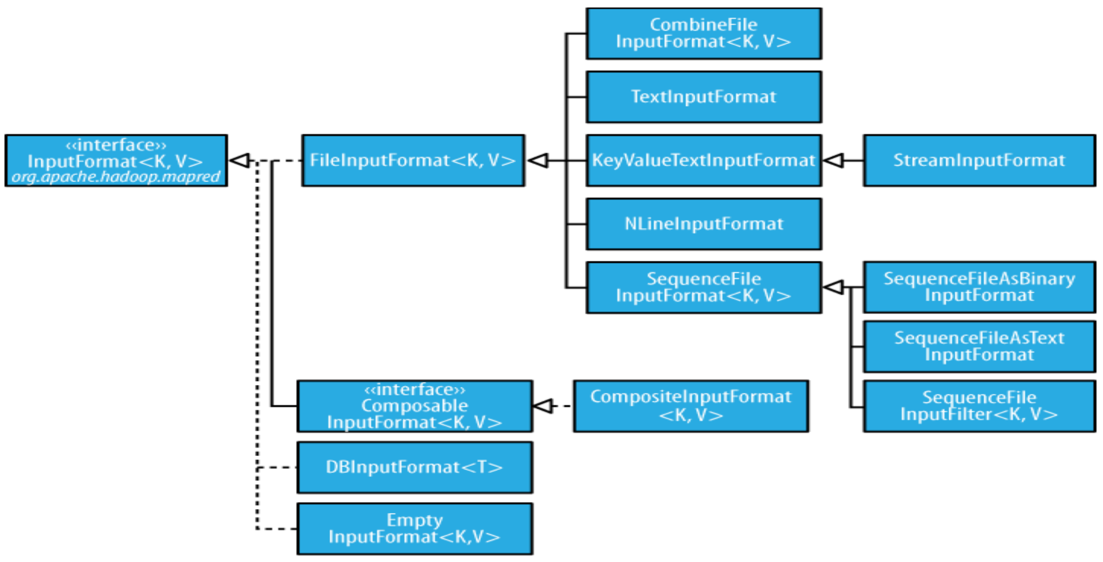

«interface» InputFormat<K, V> org.apache.hadoop.mapred
FileInputFormat<K, V>
CombineFile InputFormat<K, V>
TextInputFormat
KeyValueTextInputFormat
StreamInputFormat
NLineInputFormat
SequenceFile InputFormat<K, V>
SequenceFileAsBinary InputFormat
SequenceFileAsText InputFormat
SequenceFile InputFilter<K, V>
«interface» Composable InputFormat<K, V>
CompositeInputFormat <K, V>
DBInputFormat<T>
Empty InputFormat<K,V>

Fig: InputFormat class hierarchy

Table: ***Input path and filter properties***

Property name	Type	Default value	Description
`mapred.input.dir`	comma-separated paths	none	The input files for a job. Paths that contain commas should have those commas escaped by a backslash character. For example, the glob `{a,b}` would be escaped as `{a\,b}`.
`mapred.input.pathFilter.class`	`PathFilter` classname	none	The filter to apply to the input files for a job.

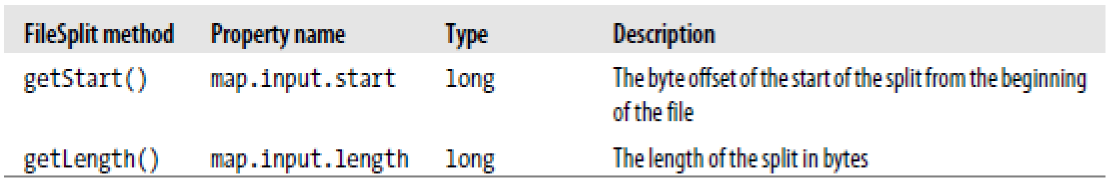
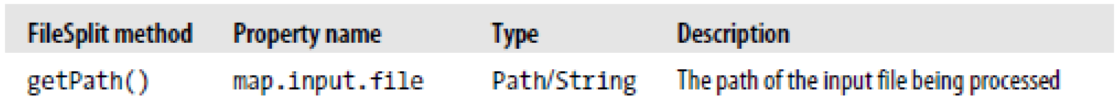
- FileInputFormat input splits: FileInputFormat splits only large files. Here "large" means larger than an HDFS block. The split size is normally the size of an HDFS block.

Table: ***Properties for controlling split size***

Property name	Type	Default value	Description
`mapred.min.split.size`	`int`	1	The smallest valid size in bytes for a file split.
`mapred.max.split.size`[a]	`long`	`Long.MAX_VALUE`, that is 9223372036854775807	The largest valid size in bytes for a file split.
`dfs.block.size`	`long`	64 MB, that is 67108864	The size of a block in HDFS in bytes.

- Preventing splitting: There are a couple of ways to ensure that an existing file is not split. The first way is to increase the minimum split size to be larger than the largest file in your system. The second is to subclass the concrete subclass of FileInputFormat that you want to use, to override the isSplitable() method to return false.
- File information in the mapper: A mapper processing a file input split can find information about the split by calling the getInputSplit() method on the Mapper's Context object.

Table: ***File split properties***

FileSplit method	Property name	Type	Description
`getStart()`	`map.input.start`	`long`	The byte offset of the start of the split from the beginning of the file
`getLength()`	`map.input.length`	`long`	The length of the split in bytes
`getPath()`	`map.input.file`	`Path/String`	The path of the input file being processed

- Processing a whole file as a record: A related requirement that sometimes crops up is for mappers to have access to the full contents of a file. The listing for WholeFileInputFormat shows a way of doing this.

Example : An InputFormat for reading a whole file as a record

```
public class WholeFileInputFormat extends FileInputFormat<NullWritable,
BytesWritable> {
@Override
protected boolean isSplitable(JobContext context, Path file) {
return false;
} }
```

WholeFileRecordReader is responsible for taking a FileSplit and converting it into a single record, with a null key and a value containing the bytes of the file.

2) Text Input

- TextInputFormat: TextInputFormat is the default InputFormat. Each record is a line of input. A file containing the following text:

  On the top of the Crumpetty Tree

  The Quangle Wangle sat,

  But his face you could not see,

  On account of his Beaver Hat.

is divided into one split of four records. The records are interpreted as the following key-value pairs:

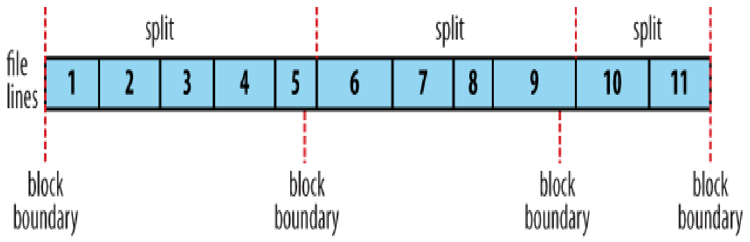
(0, On the top of the Crumpetty Tree)

(33, The Quangle Wangle sat,)

(57, But his face you could not see,)

(89, On account of his Beaver Hat.)

Fig: *Logical records and HDFS blocks for TextInputFormat*



- You can specify the separator via the mapreduce.input.keyvaluelinerecordreader.key.value.separator property. It is a tab character by default. Consider the following input file, where → represents a (horizontal) tab character:

  line1→On the top of the Crumpetty Tree

  line2→The Quangle Wangle sat,

  line3→But his face you could not see,

  line4→On account of his Beaver Hat.

Like in the TextInputFormat case, the input is in a single split comprising four records, although this time the keys are the Text sequences before the tab in each line:

(line1, On the top of the Crumpetty Tree)

(line2, The Quangle Wangle sat,)

(line3, But his face you could not see,)

(line4, On account of his Beaver Hat.)

- NLineInputFormat: N refers to the number of lines of input that each mapper receives. With N set to one, each mapper receives exactly one line of input. mapreduce.input.lineinputformat.linespermap property controls the value of N.

For example, N is two, then each split contains two lines. One mapper will receive the first two key-value pairs:

(0, On the top of the Crumpetty Tree)

(33, The Quangle Wangle sat,)

And another mapper will receive the second two key-value pairs:

(57, But his face you could not see,)

(89, On account of his Beaver Hat.)

3) Binary Input: Hadoop MapReduce is not just restricted to processing textual data—it has support for binary formats, too.

- SequenceFileInputFormat: Hadoop's sequence file format stores sequences of binary key-value pairs.
- SequenceFileAsTextInputFormat: SequenceFileAsTextInputFormat is a variant of SequenceFileInputFormat that converts the sequence file's keys and values to Text objects.
- SequenceFileAsBinaryInputFormat: SequenceFileAsBinaryInputFormat is a variant of SequenceFileInputFormat that retrieves the sequence file's keys and values as opaque binary objects.

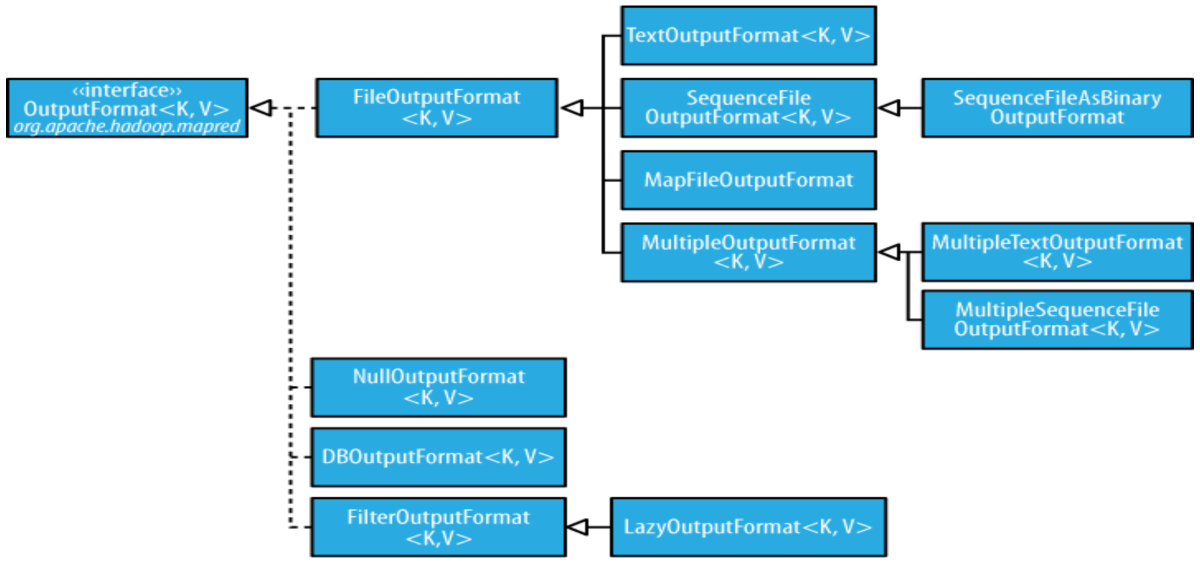
4) Multiple Inputs: Although the input to a MapReduce job may consist of multiple input files, all of the input is interpreted by a single InputFormat and a single Mapper.

The MultipleInputs class has an overloaded version of addInputPath() that doesn't take a mapper:

public static void addInputPath(Job job, Path path, Class<? extends InputFormat> inputFormatClass)

Output Formats

Figure: OutputFormat class hierarchy


«interface» OutputFormat<K, V> org.apache.hadoop.mapred
FileOutputFormat <K, V>
TextOutputFormat<K, V>
SequenceFile OutputFormat<K, V>
SequenceFileAsBinary OutputFormat
MapFileOutputFormat
MultipleOutputFormat <K, V>
MultipleTextOutputFormat <K, V>
MultipleSequenceFile OutputFormat<K, V>
NullOutputFormat <K, V>
DBOutputFormat<K, V>
FilterOutputFormat <K,V>
LazyOutputFormat<K, V>

1) Text Output: The default output format, TextOutputFormat, writes records as lines of text. Its keys
and values may be of any type, since TextOutputFormat turns them to strings by calling toString() on them.

2) Binary Output

- SequenceFileOutputFormat: As the name indicates, SequenceFileOutputFormat writes sequence files for its output. Compression is controlled via the static methods on SequenceFileOutputFormat.
- SequenceFileAsBinaryOutputFormat: SequenceFileAsBinaryOutputFormat is the counterpart to SequenceFileAsBinaryInput Format, and it writes keys and values in raw binary format into a SequenceFile container.
- MapFileOutputFormat: MapFileOutputFormat writes MapFiles as output. The keys in a MapFile must be added in order, so you need to ensure that your reducers emit keys in sorted order.

3) Multiple Outputs: FileOutputFormat and its subclasses generate a set of files in the output directory. There is one file per reducer, and files are named by the partition number: part-r-00000, partr-00001, etc. MapReduce comes with the MultipleOutputs class to help you do this.

Zero reducers: There are no partitions, as the application needs to run only map tasks.

One reducer: It can be convenient to run small jobs to combine the output of previous jobs into a single file. This should only be attempted when the amount of data is small enough to be processed comfortablyby one reducer.

- ✓ MultipleOutputs: MultipleOutputs allows you to write data to files whose names are derived from the output keys and values, or in fact from an arbitrary string.MultipleOutputs delegates to the mapper's OutputFormat, which in this example is a TextOutputFormat, but more complex set ups are possible.
- ✓ Lazy Output: FileOutputFormat subclasses will create output (part-r-nnnnn) files, even if they are
  empty. Some applications prefer that empty files not be created, which is where LazyOutputFormat helps. It is a wrapper output format that ensures that the output file is Output Formats created only when the first record is emitted for a given partition. To use it, call its setOutputFormatClass() method with the JobConf and the underlying output format.
- ✓ Database Output: The output formats for writing to relational databases and to HBase

UNIT V HADOOP RELATED TOOLS

Hbase – data model and implementations – Hbase clients – Hbase examples – praxis. Cassandra – cassandra data model – cassandra examples – cassandra clients – Hadoop integration. Pig – Grunt – pig data model – Pig Latin – developing and testing Pig Latin scripts. Hive – data types and file formats – HiveQL data definition – HiveQL data manipulation – HiveQL queries.

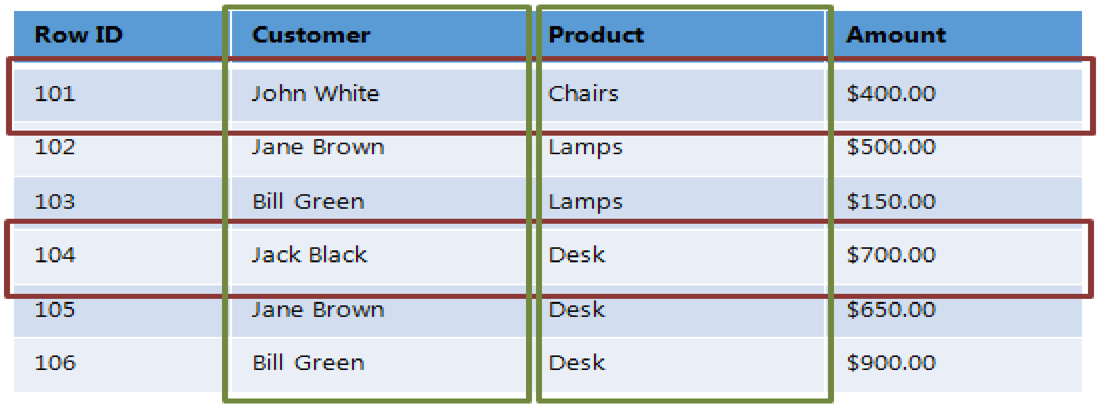
HBase

- HBASE stands for **H**adoop data **Base**
- HBase is a distributed column-oriented database built on top of HDFS.
- HBase is the Hadoop application to use when you require real-time read/write random-access to very large datasets.
- Horizontally scalable

    - – Automatic sharding
- Supports strongly consistent reads and writes
- Supports Automatic fail-over
- It has Simple Java API
- Integrated with Map/Reduce framework
- Supports Thrift, Avro and REST Web-services
- When To Use HBase
    - Good for large amounts of data
        - 100s of millions or billions of rows
        - If data is too small all the records will end up on a single node leaving the rest of the cluster idle
- When NOT to Use HBase
    - Bad for traditional RDBMs retrieval
        - Transactional applications
        - Relational Analytics
            - 'group by', 'join', and 'where column like', etc....
        - Currently bad for text-based search access

HBase is a column-oriented database that's an open-source implementation of Google's Big Table storage architecture. It can manage structured and semi-structured data and has some built-in features such as scalability, versioning, compression and garbage collection. Since its uses write-ahead logging and distributed configuration, it can provide fault-tolerance and quick recovery from individual server failures. HBase built on top of Hadoop / HDFS and the data stored in HBase can be manipulated using Hadoop's MapReduce capabilities.

Let's now take a look at how HBase (a column-oriented database) is different from some other data structures and concepts that we are familiar with Row-Oriented vs. Column-Oriented data stores. As shown below, in a row-oriented data store, a row is a unit of data that is read or written together. In a column-oriented data store, the data in a column is stored together and hence quickly retrieved.



Row ID	Customer	Product	Amount
101	John White	Chairs	$400.00
102	Jane Brown	Lamps	$500.00
103	Bill Green	Lamps	$150.00
104	Jack Black	Desk	$700.00
105	Jane Brown	Desk	$650.00
106	Bill Green	Desk	$900.00

Row-oriented data stores –

- Data is stored and retrieved one row at a time and hence could read unnecessary data if only some of the data in a row is required.
- Easy to read and write records
- Well suited for OLTP systems
- Not efficient in performing operations applicable to the entire dataset and hence aggregation is an expensive operation
- Typical compression mechanisms provide less effective results than those on column-oriented data stores

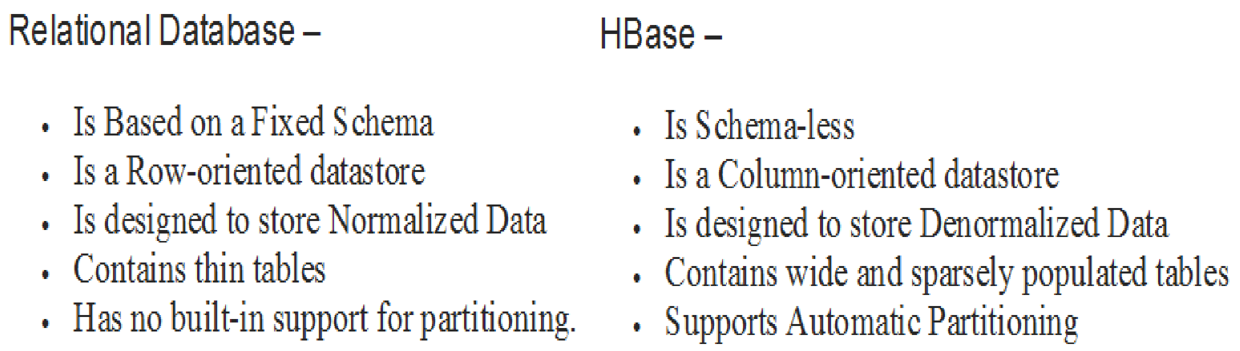
Column-oriented data stores –

- Data is stored and retrieved in columns and hence can read only relevant data if only some data is required
- Read and Write are typically slower operations
- Well suited for OLAP systems
- Can efficiently perform operations applicable to the entire dataset and hence enables aggregation over many rows and columns

- Permits high compression rates due to few distinct values in columns

Relational Databases vs. HBase

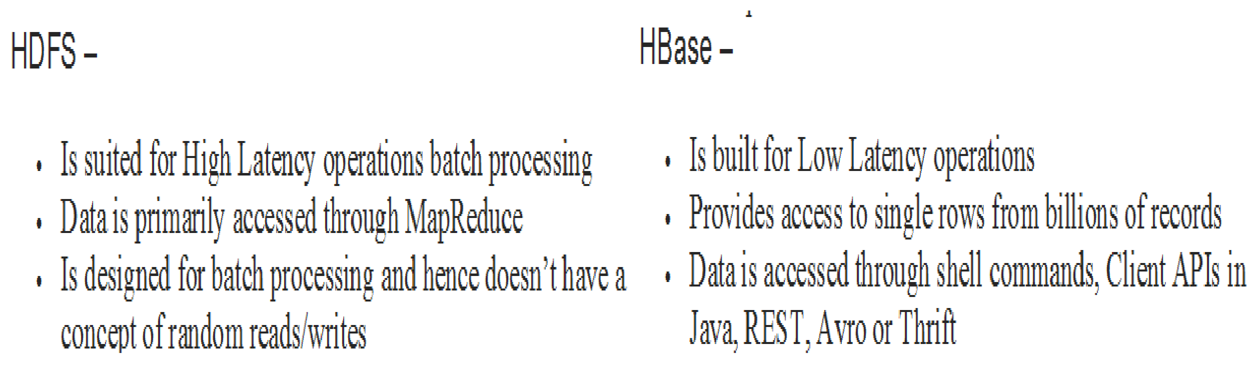
When talking of data stores, we first think of Relational Databases with structured data storage and a sophisticated query engine. However, a Relational Database incurs a big penalty to improve performance as the data size increases. HBase, on the other hand, is designed from the ground up to provide scalability and partitioning to enable efficient data structure serialization, storage and retrieval. Broadly, the differences between a Relational Database and HBase are:



Relational Database –	HBase –
• Is Based on a Fixed Schema	• Is Schema-less
• Is a Row-oriented datastore	• Is a Column-oriented datastore
• Is designed to store Normalized Data	• Is designed to store Denormalized Data
• Contains thin tables	• Contains wide and sparsely populated tables
• Has no built-in support for partitioning.	• Supports Automatic Partitioning

HDFS vs. HBase

HDFS is a distributed file system that is well suited for storing large files. It's designed to support batch processing of data but doesn't provide fast individual record lookups. HBase is built on top of HDFS and is designed to provide access to single rows of data in large tables. Overall, the differences between HDFS and HBase are

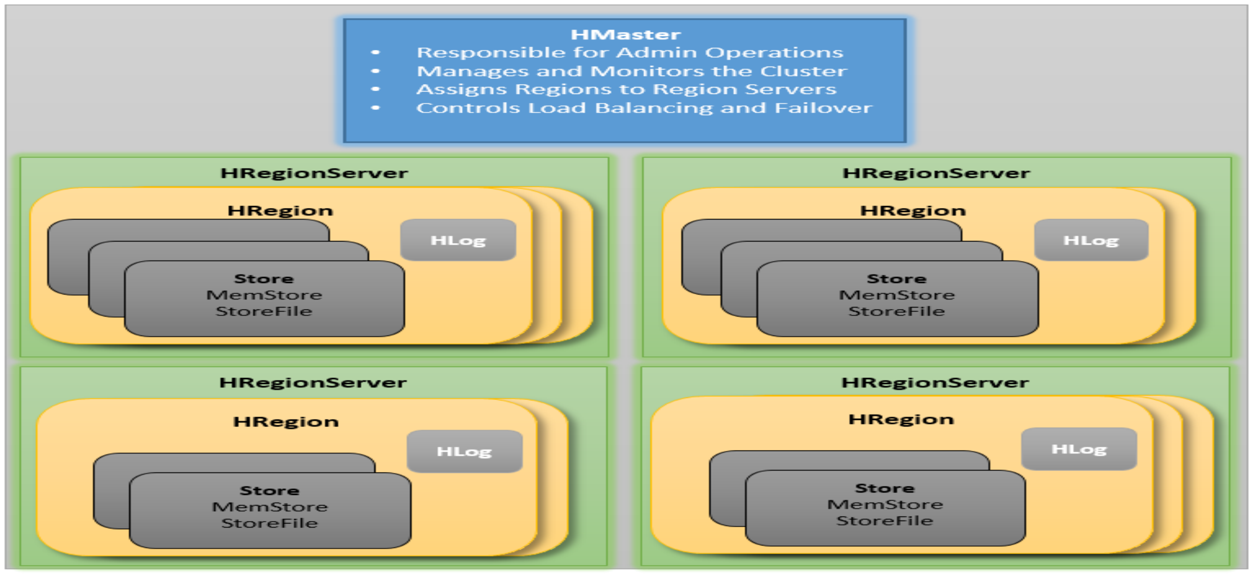


HDFS –	HBase –
• Is suited for High Latency operations batch processing	• Is built for Low Latency operations
• Data is primarily accessed through MapReduce	• Provides access to single rows from billions of records
• Is designed for batch processing and hence doesn't have a concept of random reads/writes	• Data is accessed through shell commands, Client APIs in Java, REST, Avro or Thrift

HBase Architecture

The HBase Physical Architecture consists of servers in a Master-Slave relationship as shown below. Typically, the HBase cluster has one Master node, called HMaster and multiple Region Servers called HRegionServer. Each Region Server contains multiple Regions – HRegions.

Just like in a Relational Database, data in HBase is stored in Tables and these Tables are stored in Regions. When a Table becomes too big, the Table is partitioned into multiple Regions. These Regions are assigned to Region Servers across the cluster. Each Region Server hosts roughly the same number of Regions.



The HMaster in the HBase is responsible for

- Performing Administration
- Managing and Monitoring the Cluster
- Assigning Regions to the Region Servers
- Controlling the Load Balancing and Failover

On the other hand, the HRegionServer perform the following work

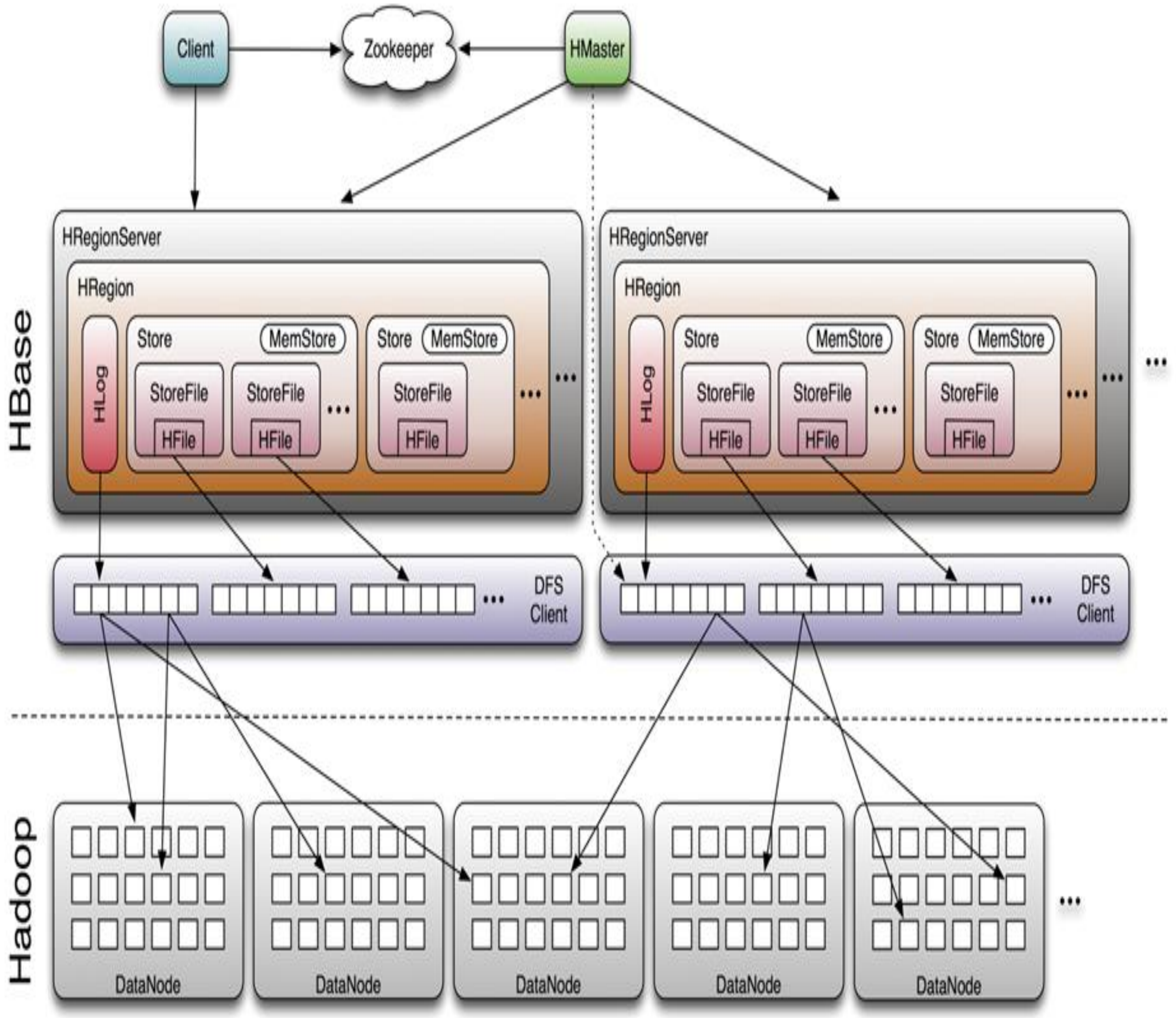
- Hosting and managing Regions

- Splitting the Regions automatically
- Handling the read/write requests
- Communicating with the Clients directly

Each Region Server contains a Write-Ahead Log (called HLog) and multiple Regions. Each Region in turn is made up of a MemStore and multiple StoreFiles (HFile). The data lives in these StoreFiles in the form of Column Families (explained below). The MemStore holds in-memory modifications to the Store (data).

The mapping of Regions to Region Server is kept in a system table called .META. When trying to read or write data from HBase, the clients read the required Region information from the .META table and directly communicate with the appropriate Region Server. Each Region is identified by the start key (inclusive) and the end key (exclusive)

HBASE detailed Architecture



You can see that HBase handles basically two kinds of file types. One is used for the write-ahead log and the other for the actual data storage. The files are primarily handled by the HRegionServer's. But in certain scenarios even the HMaster will have to perform low-level file operations. You may also notice that the actual files are in fact divided up into smaller blocks when stored within the Hadoop Distributed Filesystem (HDFS). This is also one of the areas where you can configure the system to handle larger or smaller data better. More on that later.

ZooKeeper is a high-performance coordination server for distributed applications. It exposes common services -- such as naming and configuration management, synchronization, and group services -- in a simple interface.

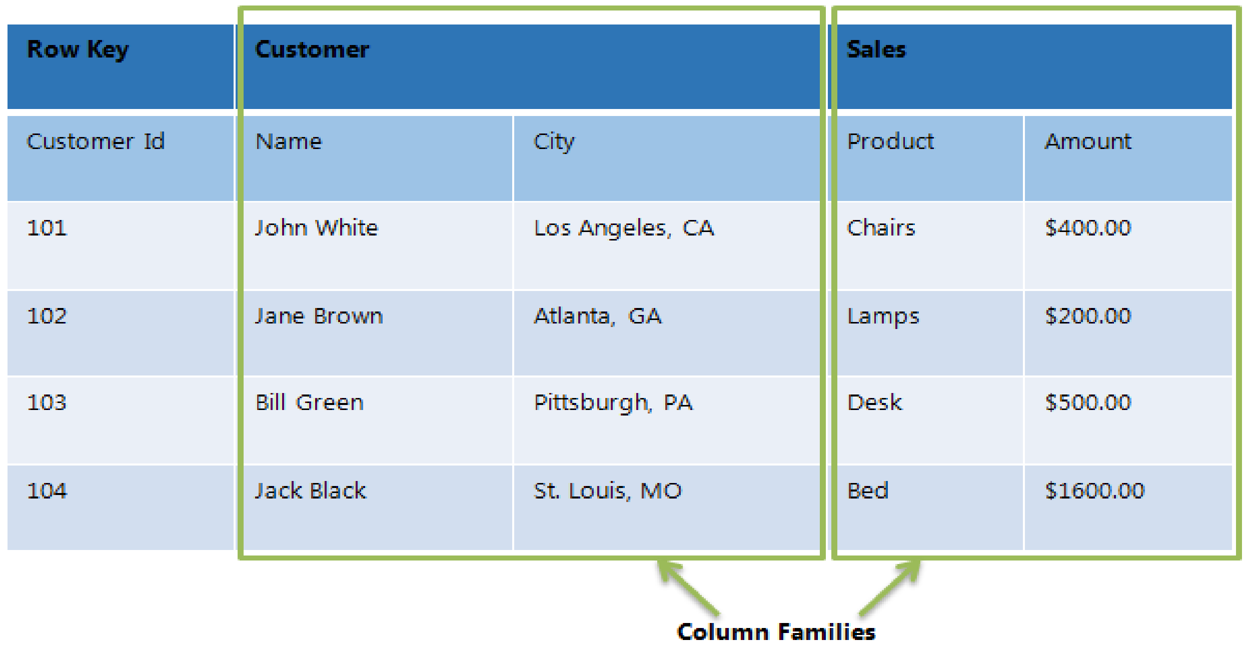
The general flow is that a new client contacts the Zookeeper quorum first to find a particular row key. It does so by retrieving the server name (i.e. host name) that hosts the -ROOT- region from Zookeeper. With that information it can query that server to get the server that hosts the .META. table. Both of these two details are cached and only looked up once. Lastly it can query the .META. server and retrieve the server that has the row the client is looking for.

Once it has been told where the row resides, i.e. in what region, it caches this information as well and contacts the HRegionServer hosting that region directly. So over time the client has a pretty complete picture of where to get rows from without needing to query the .META. server again.

Next the HRegionServer opens the region it creates a corresponding HRegion object. When the HRegion is "opened" it sets up a Store instance for each HColumnFamily for every table as defined by the user beforehand. Each of the Store instances can in turn have one or more StoreFile instances, which are lightweight wrappers around the actual storage file called HFile. A HRegion also has a MemStore and a HLog instance. We will now have a look at how they work together but also where there are exceptions to the rule.

HBase Data Model

The Data Model in HBase is designed to accommodate semi-structured data that could vary in field size, data type and columns. Additionally, the layout of the data model makes it easier to partition the data and distribute it across the cluster. The Data Model in HBase is made of different logical components such as Tables, Rows, Column Families, Columns, Cells and Versions.



Row Key	Customer		Sales	
Customer Id	Name	City	Product	Amount
101	John White	Los Angeles, CA	Chairs	$400.00
102	Jane Brown	Atlanta, GA	Lamps	$200.00
103	Bill Green	Pittsburgh, PA	Desk	$500.00
104	Jack Black	St. Louis, MO	Bed	$1600.00

Column Families

Tables – The HBase Tables are more like logical collection of rows stored in separate partitions called Regions. As shown above, every Region is then served by exactly one Region Server. The figure above shows a representation of a Table.

Rows – A row is one instance of data in a table and is identified by a *rowkey*. Rowkeys are unique in a Table and are always treated as a byte[].

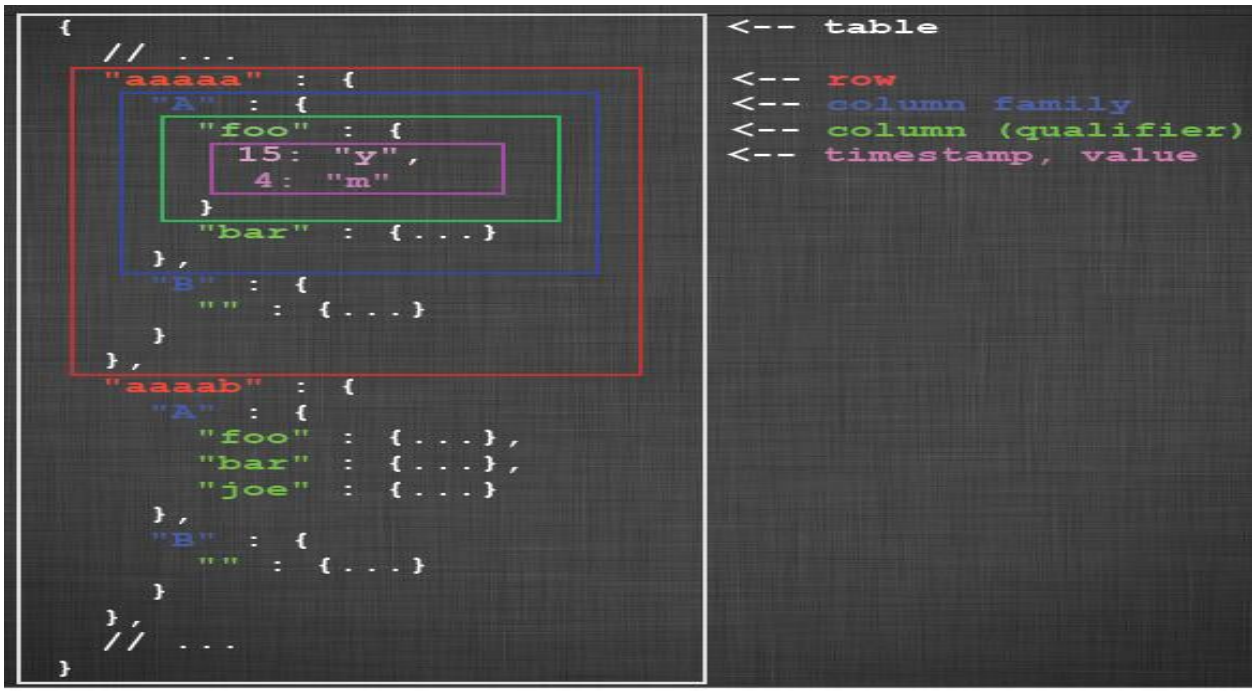
Column Families – Data in a row are grouped together as Column Families. Each Column Family has one more Columns and these Columns in a family are stored together in a low level storage file known as HFile. Column Families form the basic unit of physical storage to which certain HBase features like compression are applied. Hence it's important that proper care be taken when designing Column Families in table. The table above shows Customer and Sales Column Families. The Customer Column Family is made up 2 columns – Name and City, whereas the Sales Column Families is made up to 2 columns – Product and Amount.

Columns – A Column Family is made of one or more columns. A Column is identified by a Column Qualifier that consists of the Column Family name concatenated with the Column name using a colon – example: columnfamily:columnname. There can be multiple Columns within a Column Family and Rows within a table can have varied number of Columns.

Cell – A Cell stores data and is essentially a unique combination of *rowkey*, Column Family and the Column (Column Qualifier). The data stored in a Cell is called its value and the data type is always treated as byte[].

Version – The data stored in a cell is versioned and versions of data are identified by the timestamp. The number of versions of data retained in a column family is configurable and this value by default is 3.

In JSON format, the model is represented as



```
{                                  <-- table
  // ...
  "aaaaa" : {                      <-- row
    "A" : {                        <-- column family
      "foo" : {                    <-- column (qualifier)
        15: "y",                   <-- timestamp, value
         4: "m"
      }
      "bar" : {...}
    },
    "B" : {
      "" : {...}
    }
  },
  "aaaab" : {
    "A" : {
      "foo" : {...},
      "bar" : {...},
      "joe" : {...}
    },
    "B" : {
      "" : {...}
    }
  },
  // ...
}
```

(Table, RowKey, Family, Column, Timestamp) → Value

HBase clients

1. **REST**

HBase ships with a powerful REST server, which supports the complete client and administrative API. It also provides support for different message formats, offering many choices for a client application to communicate with the server.

REST Java client

The REST server also comes with a comprehensive Java client API. It is located in the org.apache.hadoop.hbase.rest.client package.

2. **Thrift**

Apache Thrift is written in C++, but provides schema compilers for many programming languages, including Java, C++, Perl, PHP, Python, Ruby, and more. Once you have compiled a schema, you can exchange messages transparently between systems implemented in one or more of those languages.

3. **Avro**

Apache Avro, like Thrift, provides schema compilers for many programming languages, including Java, C++, PHP, Python, Ruby, and more. Once you have compiled a schema, you can exchange messages transparently between systems implemented in one or more of those languages.

4. **Other Clients**

- **JRuby**

  The HBase Shell is an example of using a JVM-based language to access the Javabased API. It comes with the full source code, so you can use it to add the same features to your own JRuby code.
- **HBql**

  HBql adds an SQL-like syntax on top of HBase, while adding the extensions needed where HBase has unique features
- **HBase-DSL**

  This project gives you dedicated classes that help when formulating queries against an HBase cluster. Using a builder-like style, you can quickly assemble all the options and parameters necessary.
- **JPA/JPO**

  You can use, for example, DataNucleus to put a JPA/JPO access layer on top of

HBase.

- **PyHBase**

  The PyHBase project (https://github.com/hammer/pyhbase/) offers an HBase client through the Avro gateway server.
- **AsyncHBase**

  AsyncHBase offers a completely asynchronous, nonblocking, and thread-safe client to access HBase clusters. It uses the native RPC protocol to talk directly to the various servers

Cassandra

The Cassandra data store is an open source Apache project available at http://cassandra .apache.org. Cassandra originated at Facebook in 2007 to solve that company's inbox search problem, in which they had to deal with large volumes of data in a way that was difficult to scale with traditional methods.

Main features

- **Decentralized**

  Every node in the cluster has the same role. There is no single point of failure. Data is distributed across the cluster (so each node contains different data), but there is no master as every node can service any request.
- **Supports replication and multi data center replication**

  Replication strategies are configurable.[18] Cassandra is designed as a distributed system, for deployment of large numbers of nodes across multiple data centers. Key features of Cassandra's distributed architecture are specifically tailored for multiple-data center deployment, for redundancy, for failover and disaster recovery.
- **Scalability**

  Read and write throughput both increase linearly as new machines are added, with no downtime or interruption to applications.
- **Fault-tolerant**

  Data is automatically replicated to multiple nodes for fault-tolerance. Replication across multiple data centers is supported. Failed nodes can be replaced with no downtime.
- **Tunable consistency**

  Writes and reads offer a tunable level of consistency, all the way from "writes never fail" to "block for all replicas to be readable", with the quorum level in the middle.
- **MapReduce support**

Cassandra has Hadoop integration, with MapReduce support. There is support also for Apache Pig and Apache Hive.

- **Query language**

Cassandra introduces CQL (Cassandra Query Language), a SQL-like alternative to the traditional RPC interface. Language drivers are available for Java (JDBC), Python, Node.JS and Go

Why we use?

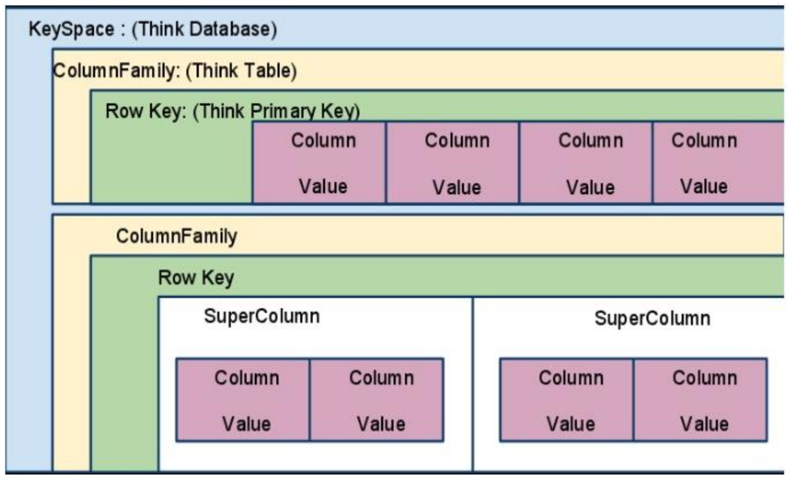
1. Quick writes
2. Fail safe
3. Quick Reporting
4. Batch Processing too, with map reduces.
5. Ease of maintenance
6. Ease of configuration
7. tune ably consistent
8. highly available
9. fault tolerant
10. The peer-to-peer design allows for high performance with linear scalability and no single points of failure
11. Decentralized databases
12. Supports 12 different client languages
13. Automatic provisioning of new nodes

Cassandra data model

Cassandra is a hybrid between a key-value and a column-oriented NoSQL databases. Key value nature is represented by a row object, in which value would be generally organized in columns. In short, cassandra uses the following terms

1. **Keyspace** can be seen as DB Schema in SQL.
2. **Column family** resembles a table in SQL world (read below this analogy is misleading)
3. **Row** has a key and as a value a set of Cassandra columns. But without relational schema corset.
4. **Column** is a triplet := (name, value, timestamp).
5. **Super column** is a tupel := (name, collection of columns).
6. **Data Types** : Validators & Comparators
7. **Indexes**

Cassandra data model is illustrated in the following figure

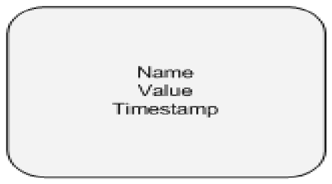


KeySpaces

KeySpaces are the largest container, with an ordered list of ColumnFamilies, similar to a database in RDMS.

Column

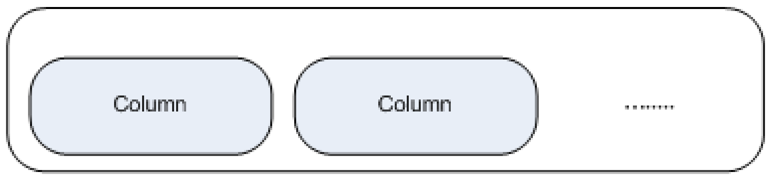
A Column is the most basic element in Cassandra: a simple tuple that contains a name, value and timestamp. All values are set by the client. That's an important consideration for the timestamp, as it means you'll need clock synchronization.



Column

SuperColumn

A SuperColumn is a column that stores an associative array of columns. You could think of it as similar to a HashMap in Java, with an identifying column (name) that stores a list of columns inside (value). The key difference between a Column and a SuperColumn is that the value of a Column is a string, where the value of a SuperColumn is a map of Columns. Note that SuperColumns have no timestamp, just a name and a value.

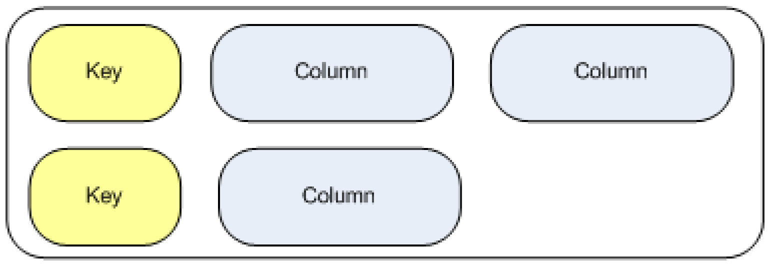


Super Column

ColumnFamily

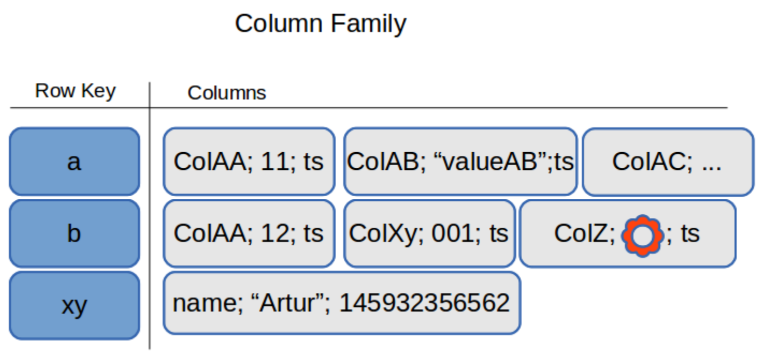
A ColumnFamily hold a number of Rows, a sorted map that matches column names to column values. A row is a set of columns, similar to the table concept from relational databases. The column family holds an ordered list of columns which you can reference by column name.

The ColumnFamily can be of two types, Standard or Super. **Standard** ColumnFamilys contain a map of normal columns,

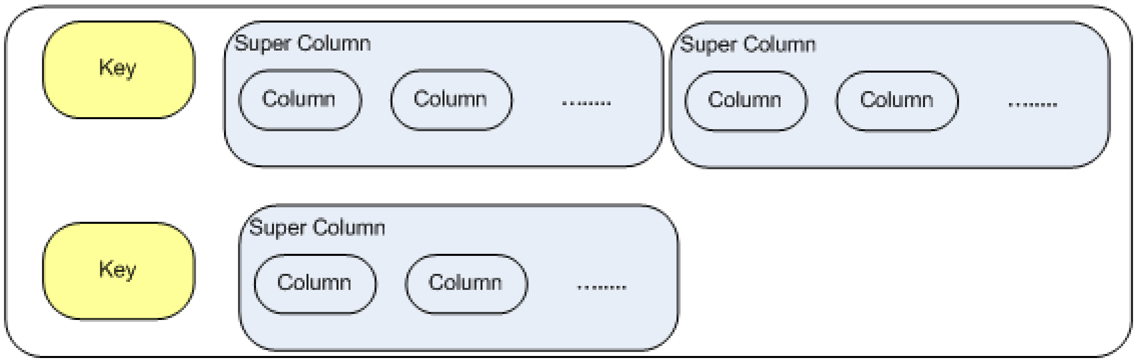


(Standard) Column Family

Example

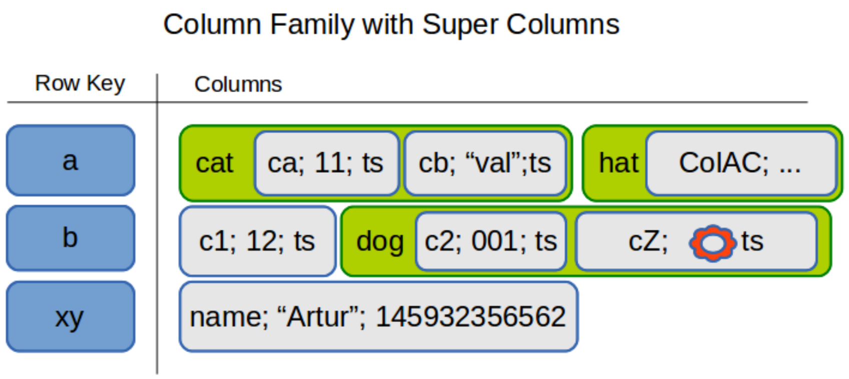


Super ColumnFamily's contain rows of SuperColumns.



(Super) Column Family

Example



Data Types

And of course there are predefined data types in cassandra, in which

- The data type of row key is called a *validator*.
- The data type for a column name is called a *comparator*.

You can assign predefined data types when you create your column family (which is recommended), but Cassandra does not require it. Internally Cassandra stores column names and values as hex byte arrays (BytesType). This is the default client encoding.

Indexes

The understanding of Indexes in Cassandra is requisite. There are two kinds of them.

- The Primary index for a column family is the index of its row keys. Each node maintains this index for the data it manages.
- The Secondary indexes in Cassandra refer to indexes on column values. Cassandra implements secondary indexes as a hidden column family

Primary index determines cluster-wide row distribution. Secondary indexes are very important for custom queries.

Differences Between RDBMS and Cassandra

1. **No Query Language:** SQL is the standard query language used in relational databases. Cassandra has no query language. It does have an API that you access through its RPC serialization mechanism, Thrift.
2. **No Referential Integrity**: Cassandra has no concept of referential integrity, and therefore has no concept of joins.
3. **Secondary Indexes:** The second column family acts as an explicit secondary index in Cassandra
4. **Sorting**: In RDBMS, you can easily change the order of records by using ORDER BY or GROUP BY in your query. There is no support for ORDER BY and GROUP BY statements in Cassandra. In Cassandra, however, sorting is treated differently; it is a design decision. Column family definitions include a CompareWith element, which dictates the order in which your rows will be sorted.
5. **Denormalization**: In the relational world, denormalization violates Codd's normal forms, and we try to avoid it. But in Cassandra, denormalization is, well, perfectly normal. It's not required if your data model is simple.
6. **Design Patterns:** Cassandra design pattern offers a Materialized View, Valueless Column, and Aggregate Key.

Cassandra Clients

5. Thrift

Thrift is the driver-level interface; it provides the API for client implementations in a wide variety of languages. Thrift was developed at Facebook and donated as an Apache project

Thrift is a code generation library for clients in C++, C#, Erlang, Haskell, Java, Objective C/Cocoa, OCaml, Perl, PHP, Python, Ruby, Smalltalk, and Squeak. Its goal is to provide an easy way to support efficient RPC calls in a wide variety of popular languages, without requiring the overhead of something like SOAP.

The design of Thrift offers the following features:

- ✓ *Language-independent types*
- ✓ *Common transport interface*
- ✓ *Protocol independence*
- ✓ *Versioning support*

6. Avro

The Apache Avro project is a data serialization and RPC system targeted as the replacement for Thrift in Cassandra.
Avro provides many features similar to those of Thrift and other data serialization and RPC mechanisms including:

- Robust data structures
- An efficient, small binary format for RPC calls
- Easy integration with dynamically typed languages such as Python, Ruby, Smalltalk, Perl, PHP, and Objective-C

Avro is the RPC and data serialization mechanism for Cassandra. It generates code that remote clients can use to interact with the database. It's well-supported in the community and has the strength of growing out of the larger and very well-known Hadoop project. It should serve Cassandra well for the foreseeable future.

7. Hector

Hector is an open source project written in Java using the MIT license. It was created by Ran Tavory of Outbrain (previously of Google) and is hosted at GitHub. It was one of the early Cassandra clients and is used in production at Outbrain. It wraps Thrift and offers JMX, connection pooling, and failover.

Hector is a well-supported and full-featured Cassandra client, with many users and an active community. It offers the following:

- ✓ High-level object-oriented API
- ✓ *Fail over support*
- ✓ *Connection pooling*
- ✓ *JMX (Java Management eXtensions) support*

8. Chirper

Chirper is a port of Twissandra to .NET, written by Chaker Nakhli. It's available under the Apache 2.0 license, and the source code is on GitHub

9. Chiton

Chiton is a Cassandra browser written by Brandon Williams that uses the Python GTK framework

10. Pelops

Pelops is a free, open source Java client written by Dominic Williams. It is similar to Hector in that it's Java-based, but it was started more recently. This has become a very popular client. Its goals include the following:

- ✓ To create a simple, easy-to-use client
- ✓ To completely separate concerns for data processing from lower-level items such
- ✓ as connection pooling
- ✓ To act as a close follower to Cassandra so that it's readily up to date

11. Kundera

Kundera is an object-relational mapping (ORM) implementation for Cassandra written using Java annotations.

12. Fauna

Ryan King of Twitter and Evan Weaver created a Ruby client for the Cassandra database called Fauna.

Pig

- ✓ Pig is a simple-to-understand data flow language used in the analysis of large data sets. Pig scripts are automatically converted into MapReduce jobs by the Pig interpreter, so you can analyze the data in a Hadoop cluster even if you aren't familiar with MapReduce.
- ✓ Used to
    - Process web log
    - Build user behavior models
    - Process images
    - Data mining

Pig is made up of two components: the first is the language itself, which is called PigLatin, and the second is a runtime environment where PigLatin programs are executed.

The Pig execution environment has two modes:

- Local mode: All scripts are run on a single machine. Hadoop MapReduce and HDFS are not required.
- Hadoop: Also called MapReduce mode, all scripts are run on a given Hadoop cluster.

Pig programs can be run in three different ways, all of them compatible with local and Hadoop mode:

1. **Pig Latin Script**: Simply a file containing Pig Latin commands, identified by the .pig suffix (for example, file.pig or myscript.pig). The commands are interpreted by Pig and executed in sequential order.
2. **Grunt shell**: Grunt is a command interpreter. You can type Pig Latin on the grunt command line and Grunt will execute the command on your behalf.
3. **Embedded**: Pig programs can be executed as part of a Java program.

Pig provides an engine for executing data flows in parallel on Hadoop. It includes a language, Pig Latin, for expressing these data flows. Pig Latin includes operators for many of the traditional data operations (join, sort, filter, etc.), as well as the ability for users to develop their own functions for reading, processing, and writing data.

- ✓ It is a large-scale data processing system
- ✓ Scripts are written in Pig Latin, a dataflow language
- ✓ Developed by Yahoo, and open source
- ✓ Pig runs on Hadoop. It makes use of both the Hadoop Distributed File System, HDFS, and Hadoop's processing system, MapReduce.

Differences between PIG vs Map Reduce

PIG is a data flow language, the key focus of Pig is manage the flow of data from input source to output store.

Pig is written specifically for managing data flow of Map reduce type of jobs. Most if not all jobs in a Pig are map reduce jobs or data movement jobs. Pig allows for custom functions to be added which can be used for processing in Pig, some default ones are like ordering, grouping, distinct, count etc.

Map/Reduce on the other hand is, it is a programming model, or framework for processing large data sets in distributed manner, using large number of computers, i.e. nodes.

PIG commands are submitted as MapReduce jobs internally.   An advantage PIG has over MapReduce is that the former is more concise. A 200 lines Java code written for MapReduce can be reduced to 10 lines of PIG code.

A disadvantage PIG has: it is bit slower as compared to MapReduce as PIG commands are translated into MapReduce prior to execution.

Pig Latin

Pig Latin has a very rich syntax. It supports operators for the following operations:

- ✓ Loading and storing of data
- ✓ Streaming data
- ✓ Filtering data
- ✓ Grouping and joining data
- ✓ Sorting data
- ✓ Combining and splitting data

Pig Latin also supports a wide variety of types, expressions, functions, diagnostic operators, macros, and file system commands.

DUMP

Dump directs the output of your script to your screen

Syntax:

dump out.txt;

LOAD : Loads data from the file system.

Syntax

```
LOAD 'data' [USING function] [AS schema];
```

'data' is the name of the file or directory, in single quotes. USING, AS are Keywords. If the USING clause is omitted, the default load function PigStorage is used. Schema- A schema using the AS keyword, enclosed in parentheses

Usage

Use the LOAD operator to load data from the file system.

Examples

Suppose we have a data file called myfile.txt. The fields are tab-delimited. The records are newline-separated.

```
1 2 3
4 2 1
8 3 4
```

In this example the default load function, PigStorage, loads data from myfile.txt to form relation A. The two LOAD statements are equivalent. Note that, because no schema is specified, the fields are not named and all fields default to type bytearray.

```
A = LOAD 'myfile.txt' USING PigStorage('\t');
DUMP A;
```

Output:

```
(1,2,3)
(4,2,1)
(8,3,4)
```

Sample Code

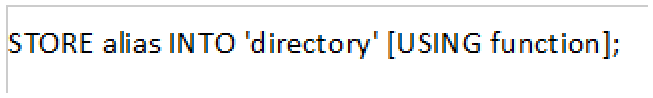
The examples are based on these Pig commands, which extract all user IDs from the

/etc/passwd file.

```
A = load 'passwd' using PigStorage(':');
B = foreach A generate $0 as id;
dump B;
store B into ‘id.txt’;
```

STORE: Stores or saves results to the file system.

Syntax



```
STORE alias INTO 'directory' [USING function];
```

Examples

In this example data is stored using PigStorage and the asterisk character (*) as the field delimiter.

```
A = LOAD 'data';
DUMP A;
```

ouptut

```
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)
```

```
STORE A INTO 'myoutput' USING PigStorage ('*');
CAT myoutput;
```

Output

```
1*2*3
4*2*1
8*3*4
4*3*3
7*2*5
8*4*3
```

STREAM : Sends data to an external script or program.

GROUP: Groups the data in one or more relations.

JOIN (inner): Performs an inner join of two or more relations based on common field values.

JOIN (outer): Performs an outer join of two relations based on common field values.

Example

Suppose we have relations A and B.

```
A = LOAD 'data1' AS (a1: int, a2: int, a3: int);

DUMP A;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)

B = LOAD 'data2' AS (b1: int, b2: int);

DUMP B;
(2,4)
(8,9)
(1,3)
(2,7)
(2,9)
(4,6)
(4,9)
```

In this example relations A and B are joined by their first fields.

```
X = JOIN A BY a1, B BY b1;

DUMP X;
(1,2,3,1,3)
(4,2,1,4,6)
(4,3,3,4,6)
(4,2,1,4,9)
(4,3,3,4,9)
(8,3,4,8,9)
(8,4,3,8,9)
```

Grunt

Grunt is Pig's interactive shell. It enables users to enter Pig Latin interactively and provides a shell for users to interact with HDFS.

In other words, it is a command interpreter. You can type Pig Latin on the grunt command line and Grunt will execute the command on your behalf.

To enter Grunt, invoke Pig with no script or command to run. Typing:

```
$ pig -x local
```

will result in the prompt:

```
grunt>
```

This gives you a Grunt shell to interact with your local filesystem. To exit Grunt you can type quit or enter Ctrl-D.

Example for entering Pig Latin Scripts in Grunt

To run Pig's Grunt shell in local mode, follow these instructions.

From your current working directory in linus, run:

```
$ pig -x local
```

The Grunt shell is invoked and you can enter commands at the prompt.

```
grunt> A = load 'passwd' using PigStorage(':');
grunt> B = foreach A generate $0 as id;
grunt> dump B;
```

Pig will not start executing the Pig Latin you enter until it sees either a store or dump.

Pig's Data Model

This includes Pig's data types, how it handles concepts such as missing data, and how you can describe your data to Pig.

Types

Pig's data types can be divided into two categories: *scalar* types, which contain a single value, and *complex* types, which contain other types.

1. **Scalar Types**

Pig's scalar types are simple types that appear in most programming languages.

- ✓ *int*

  An integer. They store a four-byte signed integer
- ✓ *long*

  A long integer. They store an eight-byte signed integer.
- ✓ *float*

  A floating-point number. Uses four bytes to store their value.
- ✓ *double*

  A double-precision floating-point number. and use eight bytes to store their value
- ✓ *chararray*

  A string or character array, and are expressed as string literals with single quotes
- ✓ *bytearray*

  A blob or array of bytes.

2. <u>Complex Types</u>

Pig has several complex data types such as maps, tuples, and bags. All of these types can contain data of any type, including other complex types. So it is possible to have a map where the value field is a bag, which contains a tuple where one of the fields is a map.

- ✓ ***Map***

  A *map* in Pig is a chararray to data element mapping, where that element can be any Pig type, including a complex type. The chararray is called a key and is used as an index to find the element, referred to as the value.

- ✓ ***Tuple***

  A *tuple* is a fixed-length, ordered collection of Pig data elements. Tuples are divided into *fields*, with each field containing one data element. These elements can be of any type—they do not all need to be the same type. A tuple is analogous to a row in SQL, with the fields being SQL columns.

- ✓ ***Bag***

  A *bag* is an unordered collection of tuples. Because it has no order, it is not possible to reference tuples in a bag by position. Like tuples, a bag can, but is not required to, have a schema associated with it. In the case of a bag, the schema describes all tuples within the bag.

- ✓ **Nulls**
  Pig includes the concept of a data element being null. Data of any type can be null. It is important to understand that in Pig the concept of null is the same as in SQL, which is completely different from the concept of null in C, Java, Python, etc. In Pig a null data element means the value is unknown.
- ✓ **Casts**
  Indicates convert one type of content to any other type.

Hive

Hive was originally an internal Facebook project which eventually tenured into a full-blown Apache project, and it was created to simplify access to MapReduce (MR) by exposing a SQL-based language for data manipulation. Hive also maintains metadata in a metastore, which is stored in a relational database, as well as this metadata contains information about what tables exist, their columns, privileges, and more. Hive is an open source data warehousing solution built on top of Hadoop, and its particular strength is in offering ad-hoc querying of data, in contrast to the compilation requirement of Pig and Cascading.

Hive is a natural starting point for more full-featured business intelligence systems which offer a user friendly interface for non-technical users.

Apache Hive supports analysis of large datasets stored in Hadoop's HDFS as well as easily compatible file systems like Amazon S3 (Simple Storage Service). Amazon S3 is a scalable, high-speed, low-cost, Web-based service designed for online backup and archiving of data as well as application programs. Hive provides SQL-like language called HiveQL while maintaining full support for map/reduce, and to accelerate queries, it provides indexes, including bitmap indexes. Apache Hive is a data warehouse infrastructure built on top of Hadoop for providing data summarization, query, as well as analysis.

Advantages of Hive

- Perfectly fits low level interface requirement of Hadoop
- Hive supports external tables and ODBC/JDBC
- Having Intelligence Optimizer
- Hive support of Table-level Partitioning to speed up the query times
- Metadata store is a big plus in the architecture that makes the lookup easy

Data Units

Hive data is organized into:

Databases: Namespaces that separate tables and other data units from naming confliction.

Tables: Homogeneous units of data, which have the same schema. An example of a table could be page_views table, where each row could comprise of the following columns (schema):

timestamp - which is of INT type that corresponds to a unix timestamp of when the page was viewed.

userid - which is of BIGINT type that identifies the user who viewed the page.

page_url - which is of STRING type that captures the location of the page.

referer_url - which is of STRING that captures the location of the page from where the user arrived at the current page.

IP - which is of STRING type that captures the IP address from where the page request was made.

Partitions: Each Table can have one or more partition Keys which determines how the data is stored. Partitions - apart from being storage units - also allow the user to efficiently identify the rows that satisfy a certain criteria. For example, a date_partition of type STRING and country_partition of type STRING. Each unique value of the partition keys defines a partition of the Table. For example all "US" data from "2009-12-23" is a partition of the page_views table. Therefore, if you run analysis on only the "US" data for 2009-12-23, you can run that query only on the relevant partition of the table thereby speeding up the analysis significantly.

Partition columns are virtual columns, they are not part of the data itself but are derived on load.

Buckets (or Clusters): Data in each partition may in turn be divided into Buckets based on the value of a hash function of some column of the Table. For example the page_views table may be bucketed by userid, which is one of the columns, other than the partitions columns, of the page_view table. These can be used to efficiently sample the data.

Hive Data Types

Hive Support two types of data type formats

1. Primitive data type
2. Collection data type

13. Primitive Data Types

Data Type	Size	Example
TINYINT	1 Byte	10, -10
SMALLINT	2 Byte	10, -10
INT	4 Byte	10, -10
BIGINT	8 Byte	10, -10
FLOAT	Single precision float	10.8932
DOUBLE	Double precision float	10.8932
BOOLEAN	Boolean true or false	TRUE
STRING	Sequence of characters	'Sample string'
TIMESTAMP	Integer, float or string values.	129357385 8929245.879395 '2013-01-01 12:00:00.123456789'
BINARY	Array of bytes	

- TINYINT, SMALLINT, INT, BIGINT are four integer data types with only differences in their size.

- FLOAT and DOUBLE are two floating point data types. BOOLEAN is to store true or false.

- STRING is to store character strings. Note that, in hive, we do not specify length for STRING like in other databases. It's more flexible and variable in length.

- TIMESTAMP can be an integer which is interpreted as seconds since UNIX epoch time. It may be a float where number after decimal is nanosecond. It may be string which is interpreted
- according to the JDBC date string format i.e. YYYY-MM-DD hh:mm:ss.fffffffff. Time component is interpreted as UTC time.

- BINARY is used to place raw bytes which will not be interpreted by hive. It is suitable for binary data.

14. Collection Data Types

1. STRUCT
2. MAP
3. ARRAY

Type	Description	Literal syntax examples
STRUCT	Analogous to a C struct or an "object." Fields can be accessed using the "dot" notation. For example, if a column name is of type STRUCT {first STRING; last STRING}, then the first name field can be referenced using name.first.	struct('John', 'Doe')
MAP	A collection of key-value tuples, where the fields are accessed using array notation (e.g., ['key']). For example, if a column name is of type MAP with key→value pairs 'first'→'John' and 'last'→'Doe', then the last name can be referenced using name['last'].	map('first', 'John', 'last', 'Doe')
ARRAY	Ordered sequences of the *same* type that are indexable using zero-based integers. For example, if a column name is of type ARRAY of strings with the value ['John', 'Doe'], then the second element can be referenced using name[1].	array('John', 'Doe')

Hive File formats

Hive supports all the Hadoop file formats, plus Thrift encoding, as well as supporting pluggable SerDe (serializer/deserializer) classes to support custom formats.

There are several file formats supported by Hive.

TEXTFILE is the easiest to use, but the least space efficient.

SEQUENCEFILE format is more space efficient.

MAPFILE which adds an index to a SEQUENCEFILE for faster retrieval of particular records.

Hive defaults to the following record and field delimiters, all of which are non-printable control characters and all of which can be customized.

Hive's default record and field delimiters

Delimiter	Description
\n	For text files, each line is a record, so the line feed character separates records.
^A ("control" A)	Separates all fields (columns). Written using the octal code \001 when explicitly specified in CREATE TABLE statements.
^B	Separate the elements in an ARRAY or STRUCT, or the key-value pairs in a MAP. Written using the octal code \002 when explicitly specified in CREATE TABLE statements.
^C	Separate the key from the corresponding value in MAP key-value pairs. Written using the octal code \003 when explicitly specified in CREATE TABLE statements.

Let us take an example to understand it. I am assuming an employee table with below structure in hive.

```
CREATE TABLE employees      (name          STRING,
                             salary        FLOAT,
                             subordinates  ARRAY<STRING>,
                             deductions    MAP<STRING, FLOAT>
                             address       STRUCT<stree:STRING,
                                                  city:STRING,
                                                  zip:INT>
                            )
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002'
MAP KEYS TERMINATED BY '\003'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

Note that \001 is an octal code for ^A, \002 is ^B and \003 is ^C. for further explanation, we will use text instead of octal code. Let's assume one record as shown in below table.

Name	Salary	Subordinates	Deductions		Address
Prashant	10000	Purna Dash	PF	750	Marathalli
		Sudarshan Raju	IT	1000	Bangalore
					560037

So, we have an employee named as Prashant. Salary is 10000. He has two subordinates Purna and Sudarshan. From his monthly salary, there are two deductions i.e. 750 for PF and 1000 as income tax. His address is Steet= Marathalli, City is Bangalore and zip code is 560037.

Let's build this record in hive table format as described by hive table structure above.

Fields are terminated by **^A**. We have five fields Name, Salary, Subordinates, Deductions and Address. You might see it like below.

Name**^A**Salary**^A**Subordinates**^A**Deductions**^A** Address

Let's replace it with actual values for first two fields because they are straight.

Prashant**^A**10000**^A**Subordinates**^A**Deductions**^A** Address

Now, we have two subordinates and we know that collection items will be delimited by **^B** giving below record. Same will apply to address parts.

Prashant**^A**10000**^A**Purna Dash**^B**Sudarshan Raju **^A**Deductions**^A**
Marathalli**^B**Bangalore**^B**560037

Deduction is bit complex than others. You need to understand that deductions has two collection items (PF**=**750 and IT**=**1000) and each of those two collection items is a MAP. MAP will be delimited by **^C** (PF**^C**750 and IT**^C**1000) then collection will be delimited by **^B** giving below record.

Prashant**^A**10000**^A**Purna Dash**^B**Sudarshan Raju **^A** PF**^C**750**^B** IT**^C**1000**^A**
Marathalli**^B**Bangalore**^B**560037

There are other file formats but for now you can consider that's how data in Hive table will be stored. It's a series of records in the above format where each record is terminated by a new line as specified in LINES TERMINATED BY '\n' clause of create table.

All delimiters used in above example are default delimiters for hive. If you do not specify them, default values will be assumed by hive. You are free to change these values for each table you have.

HiveQL

HiveQL is the Hive query language

Hadoop is an open source framework for the distributed processing of large amounts of data across a cluster. It relies upon the MapReduce paradigm to reduce complex tasks into smaller parallel tasks that can be executed concurrently across multiple machines. However, writing MapReduce tasks on top of Hadoop for processing data is not for everyone since it requires learning a new framework and a new programming paradigm altogether. What is needed is an easy-to-use abstraction on top of Hadoop that allows people not familiar with it to use its capabilities as easily.

Hive aims to solve this problem by offering an SQL-like interface, called HiveQL, on top of Hadoop. Hive achieves this task by converting queries written in HiveQL into MapReduce tasks that are then run across the Hadoop cluster to fetch the desired results

Hive is best suited for batch processing large amounts of data (such as in data warehousing) but is not ideally suitable as a routine transactional database because of its slow response times (it needs to fetch data from across a cluster).

A common task for which Hive is used is the processing of logs of web servers. These logs have a regular structure and hence can be readily converted into a format that Hive can understand and process

Hive query language (HiveQL) supports SQL features like CREATE tables, DROP tables, SELECT ... FROM ... WHERE clauses, Joins (inner, left outer, right outer and outer joins), Cartesian products, GROUP BY, SORT BY, aggregations, union and many useful functions on primitive as well as complex data types. Metadata browsing features such as list databases, tables and so on are also provided. HiveQL does have limitations compared with traditional RDBMS SQL. HiveQL allows creation of new tables in accordance with partitions(Each table can have one or more partitions in Hive) as well as buckets (The data in partitions is further distributed as buckets)and allows insertion of data in single or multiple tables but does not allow deletion or updating of data

HiveQL: Data Definition

First open the hive console by typing:

```
$ hive
```

Once the hive console is opened, like

```
hive>
```

you need to run the query to create the table.

3. **Create and Show database**

They are very useful for larger clusters with multiple teams and users, as a way of avoiding table name collisions. It’s also common to use databases to organize production tables into logical groups. If you don’t specify a database, the default database is used.

```
hive> CREATE DATABASE IF NOT EXISTS financials;
```

At any time, you can see the databases that already exist as follows:

```
hive> SHOW DATABASES;
```

output is

default

financials

```
hive> CREATE DATABASE human_resources;
```

```
hive> SHOW DATABASES;
```

output is
default
financials
human_resources

2. DESCRIBE database

- shows the directory location for the database.

```
hive> DESCRIBE DATABASE financials;
```

output is
hdfs://master-server/user/hive/warehouse/financials.db

15. USE database

The USE command sets a database as your working database, analogous to changing working directories in a filesystem

```
hive> USE financials;
```

16. DROP database

you can drop a database:

```
hive> DROP DATABASE IF EXISTS financials;
```

The IF EXISTS is optional and suppresses warnings if financials doesn’t exist.

17. Alter Database

You can set key-value pairs in the DBPROPERTIES associated with a database using the ALTER DATABASE command. No other metadata about the database can be changed,including its name and directory location:

```
hive> ALTER DATABASE financials SET DBPROPERTIES ('edited-by' = 'active steps');
```

18. Create Tables

The CREATE TABLE statement follows SQL conventions, but Hive's version offers sig- nificant extensions to support a wide range of flexibility where the data files for tables are stored, the formats used, etc.

- **Managed Tables**

✓ The tables we have created so far are called managed tables or sometimes called internal tables, because Hive controls the lifecycle of their data. As we've seen,Hive stores the data for these tables in subdirectory under the directory defined by hive.metastore.warehouse.dir (e.g., /user/hive/warehouse), by default.

✓ When we drop a managed table, Hive deletes the data in the table.

✓ Managed tables are less convenient for sharing with other tools

- **External Tables**

```
CREATE EXTERNAL TABLE IF NOT EXISTS stocks (
        exchange STRING,
        symbol STRING,
        ymd STRING,
        price_open FLOAT,
        price_high FLOAT,
        price_low FLOAT,
        price_close FLOAT,
        volume INT,
        price_adj_close FLOAT
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/data/stocks/';
```

The EXTERNAL keyword tells Hive this table is external and the LOCATION … clause is required to tell Hive where it's located. Because it's external

Partitioned, Managed Tables

Partitioned tables help to organize data in a logical fashion, such as hierarchically. Example:Our HR people often run queries with WHERE clauses that restrict the results to a particular country or to a particular first-level subdivision (e.g., state in the United States or province in Canada).

we have to use address.state to project the value inside the address. So, let’s partition the data first by country and then by state:

```
CREATE TABLE IF NOT EXISTS mydb.employees (
 name STRING,
 salary FLOAT,
 subordinates ARRAY<STRING>,
 deductions MAP<STRING, FLOAT>,
 address STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
)
PARTITIONED BY (country STRING, state STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002'
MAP KEYS TERMINATED BY '\003'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

Partitioning tables changes how Hive structures the data storage. If we create this table in the mydb database, there will still be an employees directory for the table:

```
LOAD DATA LOCAL INPATH '/path/to/employees.txt'
INTO TABLE employees
PARTITION (country = 'US', state = 'IL');
hdfs://master_server/user/hive/warehouse/mydb.db/employees
```

Once created, the partition keys (country and state, in this case) behave like regular columns.

```
hive> SHOW PARTITIONS employees;
```

output is

```
OK
country=US/state=IL
Time taken: 0.145 seconds
```

19. Dropping Tables

The familiar DROP TABLE command from SQL is supported:

```
DROP TABLE IF EXISTS employees;
```

HiveQL: Data Manipulation

1. **Loading Data into Managed Tables**

Create stocks table

```
CREATE EXTERNAL TABLE IF NOT EXISTS stocks (
        exchange STRING,
        symbol STRING,
        ymd STRING,
        price_open FLOAT,
        price_high FLOAT,
        price_low FLOAT,
        price_close FLOAT,
        volume INT,
        price_adj_close FLOAT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION '/data/stocks/';
Queries on Sotck Data Set
```

Load the stocks

```
LOAD DATA LOCAL INPATH '/path/to/employees.txt'
INTO TABLE stocks
PARTITION (exchange = 'NASDAQ', symbol = 'AAPL');
```

This command will first create the directory for the partition, if it doesn't already exist, then copy the data to it.

2. **Inserting Data into Tables from Queries**

```
INSERT OVERWRITE TABLE employees PARTITION (country = 'US', state = 'OR')
```

With OVERWRITE, any previous contents of the partition are replaced. If you drop the keyword OVERWRITE or replace it with INTO, Hive appends the data rather than replaces it.

HiveQL queries

1. **SELECT … FROM Clauses**

SELECT is the projection operator in SQL. The FROM clause identifies from which table, view, or nested query we select records

Create employees

```
CREATE EXTERNAL TABLE employees (
             name STRING,
            salary FLOAT,
            subordinates ARRAY<STRING>,
            deductions MAP<STRING, FLOAT>,
            address STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002'
MAP KEYS TERMINATED BY '\003'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
LOCATION '/data/employees';
```

Load data

```
LOAD DATA LOCAL INPATH '/path/to/employees.txt'
INTO TABLE employees
PARTITION (country = 'US', state = 'IL');
```

Data in employee.txt is assumed as

```
John Doe 100000.0 ["Mary Smith","Todd Jones"] {"Federal Taxes":0.2,"State Taxes":0.05,"Insurance":0
Mary Smith 80000.0 ["Bill King"] {"Federal Taxes":0.2,"State Taxes":0.05,"Insurance":0.1} {"street"
Todd Jones 70000.0 [] {"Federal Taxes":0.15,"State Taxes":0.03,"Insurance":0.1} {"street":"200 Chic
Bill King 60000.0 [] {"Federal Taxes":0.15,"State Taxes":0.03,"Insurance":0.1} {"street":"300 Obscu
Boss Man 200000.0 ["John Doe","Fred Finance"] {"Federal Taxes":0.3,"State Taxes":0.07,"Insurance":0
Fred Finance 150000.0 ["Stacy Accountant"] {"Federal Taxes":0.3,"State Taxes":0.07,"Insurance":0.05
Stacy Accountant 60000.0 [] {"Federal Taxes":0.15,"State Taxes":0.03,"Insurance":0.1} {"street":"30
```

Select data

```
hive> SELECT name, salary FROM employees;
```

output is

```
John Doe
100000.0
Mary Smith 80000.0
Todd Jones 70000.0
Bill King 60000.0
```

When you select columns that are one of the collection types, Hive uses JSON (Java- Script Object Notation) syntax for the output. First, let's select the subordinates, an ARRAY, where a comma-separated list surrounded with [...] is used.

```
hive> SELECT name, subordinates FROM employees;
```

output is

```
John Doe
["Mary Smith","Todd Jones"]
Mary Smith ["Bill King"]
Todd Jones []
Bill King []
```

The deductions is a MAP, where the JSON representation for maps is used, namely a comma-separated list of key:value pairs, surrounded with {...}:

```
hive> SELECT name, deductions FROM employees;
```

output is

```
John Doe
{"Federal Taxes":0.2,"State Taxes":0.05,"Insurance":0.1}
Mary Smith {"Federal Taxes":0.2,"State Taxes":0.05,"Insurance":0.1}
Todd Jones {"Federal Taxes":0.15,"State Taxes":0.03,"Insurance":0.1}
Bill King {"Federal Taxes":0.15,"State Taxes":0.03,"Insurance":0.1}
```

Finally, the address is a STRUCT, which is also written using the JSON map format:

```
hive> SELECT name, address FROM employees;
```

output is

```
John Doe
{"street":"1 Michigan Ave.","city":"Chicago","state":"IL","zip":60600}
Mary Smith {"street":"100 Ontario St.","city":"Chicago","state":"IL","zip":60601}
Todd Jones {"street":"200 Chicago Ave.","city":"Oak Park","state":"IL","zip":60700}
Bill King {"street":"300 Obscure Dr.","city":"Obscuria","state":"IL","zip":60100}
```